

国际信息工程先进技术译丛

Springer

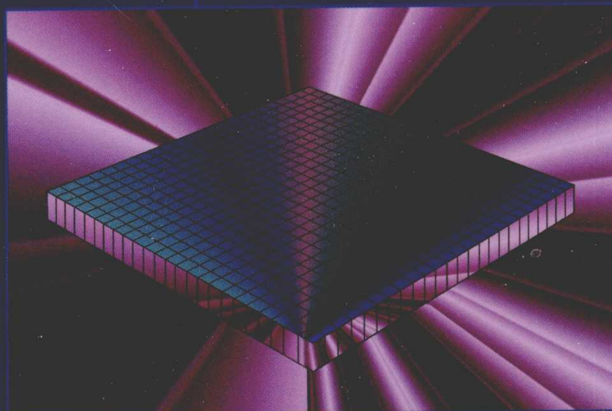
# 嵌入式系统设计——嵌入式信息物理系统基础（原书第2版）

**Embedded System Design Embedded Systems Foundations of Cyber-Physical Systems 2nd Edition**

（德） Peter Marwedel 著

何宗彬 任国栋 等译





机械工业出版社
CHINA MACHINE PRESS

国际信息工程先进技术译丛

# 嵌入式系统设计——嵌入式信息物理系统基础

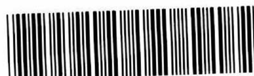
## （原书第2版）

（德）Peter Marwedel 著
何宗彬 任国栋 等译






机 械 工 业 出 版 社



本书针对近年来电子与通信技术的发展对嵌入式系统的需求，从总体上介绍了嵌入式系统的设计模式与方法，从系统的规范与建模、嵌入式硬件、嵌入式操作系统、系统的评估与验证、应用程序的实现与优化等方面，对信息—物理系统的嵌入式设计进行了讲述。透过本书，读者可以学习到更多关于嵌入式领域的前沿知识与设计方法，也可以进一步巩固嵌入式系统知识。本书对工程实践也有着较强的指导意义。

本书可以作为工程师的嵌入式学习资料，也可以作为本科、硕士和研究人员的参考书，对当前的课程教学能起到很好的补充作用。



**图书在版编目（CIP）数据**

嵌入式系统设计——嵌入式信息物理系统基础（原书第 2 版）/（德）马维戴尔（Marwedel, P.）著；何宗彬等译．—北京：机械工业出版社，2013.2

（国际信息工程先进技术译丛）

书名原文：Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems 2nd Edition

ISBN 978-7-111-41255-7

Ⅰ.①嵌… Ⅱ.①马…②何… Ⅲ.①微型计算机－系统设计 Ⅳ.①TP360.21

中国版本图书馆 CIP 数据核字（2013）第 015318 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）
策划编辑：顾　谦　责任编辑：顾　谦
版式设计：霍永明　责任校对：刘雅娜　张晓蓉
封面设计：马精明　责任印制：邓　博
北京机工印刷厂印刷（三河市南杨庄国丰装订厂装订）
2013 年 3 月第 1 版第 1 次印刷
169mm×239mm · 18.75 印张 · 381 千字
0 001—3 000 册
标准书号：ISBN 978-7-111-41255-7
定价：79.90 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

| 电话服务 | 网络服务 |
|---|---|
| 社服务中心：(010)88361066 | 教材网：http://www.cmpedu.com |
| 销售一部：(010)68326294 | 机工官网：http://www.cmpbook.com |
| 销售二部：(010)88379649 | 机工官博：http://weibo.com/cmp1952 |
| 读者购书热线：(010)88379203 | **封面无防伪标均为盗版** |

# 译　者　序

随着电子技术、通信技术等的飞速发展，嵌入式系统已经广泛地应用在工业控制、通信、航空航天、消费电子产品等领域，其所带来的价值不可估量。随着时间推移，嵌入式系统的需求量呈现指数增长，并且应用范围不断扩大，同时对系统的复杂性、稳定性、安全性以及关键性的要求也日益提高。嵌入式系统如何满足这种需求，怎样提高嵌入式软件的开发效率，怎样以最短的时间开发出最令人满意的、高效可靠的嵌入式软件成为了摆在人们面前的问题。

本书以全面而整体的视角，重新审视嵌入式系统，全面总结了嵌入式系统中常见的以及关键的设计模式及设计方法。这些模式及方法广泛应用于嵌入式系统或嵌入式软件中。本书还提出了很多新颖的设计模式，为嵌入式系统开发者提供了强有力的工具。通过这些模式，开发者可以用最短的时间设计出性能好、稳定性强、安全性高的嵌入式系统或软件，而且也能为系统日后的升级维护打下坚实的设计基础。读者能够从本书系统地掌握嵌入式系统的设计模式进行系统的开发。本书针对嵌入式系统中从硬件设计到操作系统选择、方案的评估及验证、应用程序的验证、系统的优化及测试出发，对嵌入式系统设计方面的知识进行了详细阐述，本书的读者可以从这些设计原则中进行良好的嵌入式架构设计。

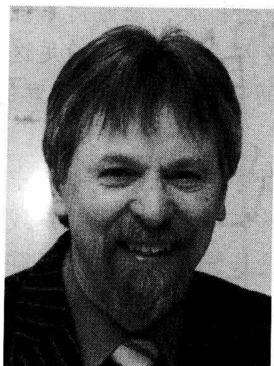
诚然，本书不是一本简简单单的工程类书籍，读完本书后某些方法并不能马上用于具体的工程实践中，但是本书所介绍的嵌入式开发思想却时时刻刻影响着所有阅读过本书的嵌入式开发人员。

本书在翻译的过程中得到了很多人的帮助和鼓励，在此感谢机械工业出版社编辑在本书翻译过程中给予的帮助，还要感谢李兵、李华、黄志登、杨建华、王耀、刘悦、王晨、李亚东、刘葆、崔波、魏玉冰、何朝荣、张凯、孙江涛、何钰、殷凤梅等人对本书部分内容翻译方面给予的指导以及对本书部分内容的翻译。由于时间关系，虽然尽最大的努力翻译，但是译文中难免有疏漏和错误之处，恳请读者批评指正。

译　者

# 关 于 作 者

**Peter Marwedel**



Peter Marwedel 出生在德国汉堡（Hamburg），他于1974年在德国基尔大学（The University of Kiel）获得物理学博士学位。1974～1989年，Peter Marwedel 一直在该校的计算科学与应用数学学院工作。从1989年起，他受聘成为德国多特蒙德工业大学（TU Dortmund）教授，而后一直在该校计算机科学学院的嵌入式系统研究方面担任领军人物，同时他也在德国一家专门从事技术转让的公司 ICD e. V. 担任重要职务。Marwedel 教授于1985/1986年在帕德伯恩大学（The University of Paderborn）担任客座教授，1995年在加州大学欧文分校（The University of California at Irvine）担任客座教授。1992～1995年，Marwedel 教授担任多特蒙德工业大学计算机科学学院院长。Marwedel 教授在 DATE 会议的成功开展方面起到了非常积极的作用，他还发起了 SCOPES 和 Map2MPSoCs 的一系列研讨会。他在1975年开始了对高层综合方面的研究，尤其是在 VLIW 的综合方面。而后，他的研究领域扩展到了嵌入式处理器（特别是处理器的可重定向性）。Marwedel 教授研究项目还包括对处理器自测试程序的综合，以及 Levi 多媒体单元的设计（参考 http://ls12-www.cs.tu-dortmund.de/teaching/download/index.html）。Marwedel 教授是 ArtistDesign 小组负责人，ArtistDesign 是欧洲一个研究嵌入式与实时系统设计艺术的组织，他领导着嵌入式系统能耗方面相关的一些项目，尤其关注嵌入式系统中内存架构以及时序的可预测性。Marwedel 教授在2003年获得其所在大学的教学成就奖。

Marwedel 教授是 IEEE 会员、DATE 会员、ACM 高级会员以及 Gesellschaft für Informatik（GI）会员。

Marwedel 教授已经结婚并有两个女儿与一个儿子，他的爱好包含轨道建模以及摄影。

E-mail：peter.marwedel@tu-dortmund.de。

网站：http://ls12-www.cs.tu-dortmund.de/~marwedel。

# 原书前言

## 定义与范围

直到20世纪80年代末期，信息处理仍然依赖于大型计算机及海量的磁带式存储介质。在90年代，信息处理开始转向使用个人计算机（PC）。随着产品小型化趋势的发展，大多数的信息处理设备将使用小型的便携式计算机，大型产品将集成多个信息处理设备。PC上的信息处理较常见，但其实信息处理也存在于多种大型产品中，如电信设备。一般来讲，科技产品必须以先进的技术来吸引顾客。在科技发达国家的汽车、摄像机、电视机、手机等领域，如果产品中没有嵌入式计算机，产品将很难畅销。基于此，并根据一些预测（参见［National Research Council，2001］），信息与通信技术（Information and Communication Technologies，ICT）的未来特征可以用如下的一些术语来概括：

1）普及计算［Weiser，2003］；

2）普适计算［Hansmann，2001］，［Burkhardt，2001］；

3）环境智能［Koninklijke Philips Electronics N. V.，2003］，［Marzano and Aarts，2003］；

4）消失的计算机［Weiser，2003］；

5）后PC时代。

术语1与2都反映了计算（与通信）无处不在的事实，信息将随时随地触手可及。这些关于未来的预测，也包括我们的日常生活将进入普遍计算的时代。作为环境智能，它们的重点之一就是未来的智能建筑。其实术语1~3仅有些许的差异：普适计算侧重于无时无处不在的信息分享；普及计算侧重于如何对现有信息进行利用。术语4表明了处理器与软件将更广泛地应用于很多小系统中，很多时候它们都不是直接可见的。术语“后PC时代”（Post-PC era）则是指硬件平台中基于标准架构的PC将会减少。

下一代ICT系统需要两项关键技术：

1）嵌入式系统（Embedded Systems）；

2）通信技术（Communication Technologies）。

图0.1展示了嵌入式系统与通信技术如何影响普适计算。

举例来说，一些嵌入式的普适计算设备，在使用诸如网络化的基本通信技术时，必须满足嵌入式系统关于实时性与可靠性的要求。除部分章节有所提及处，关于通信技术的细节并不在本书的讨论范围，读者需要另行参考其他著作。那么，究

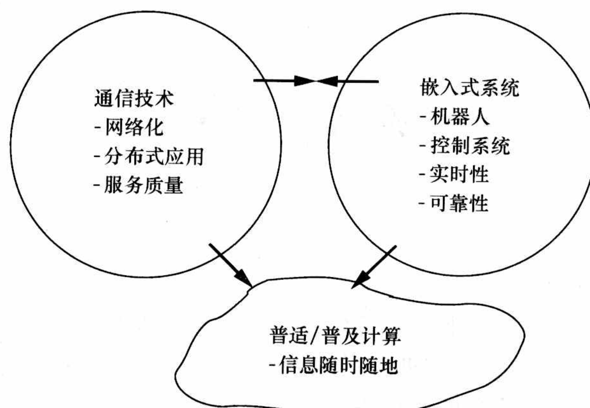



图0.1 嵌入式系统对普适/普及计算的影响（© European Commission）

竟什么是“嵌入式系统”？它们可以有如下定义［Marwedel，2003］：

**定义：嵌入式系统是集成在产品中的信息处理单元。**

如汽车、火车、飞机、电信及制造设备等包含嵌入式系统的设备，它们有很多共性，如对实时性的要求、对可靠性以及效率的要求等。在这些系统中，与物理环境及物理系统的连接尤为重要，下面的引用也对此进行了强调［Lee，2006］：

“嵌入式软件是基于物理处理器的一种软件，它用于在计算系统中管理时间与并发性”。

以上引用可以被用来定义“嵌入式软件”这一术语，如果将其中的“软件”换成“系统”，它也可以扩展为对嵌入式系统的定义。如果加以强调其与物理系统的连接，则可以引申出“信息—物理系统”（Cyber-Physical System，CPS或cy-phy系统）。cy-phy系统的定义如下：

**定义：“信息—物理系统是计算与物理过程的集成”**［Lee，2007］。

这个新的术语强调了诸如时间、能量、空间与物理工程的关系。在大部分软件都运行在PC上的环境中，这种关系常常被忽略，那么现在对其进行强调，就有着重要的工程意义。对于信息—物理系统，在系统建模时，需要同时考虑物理环境的模型。从这种意义上讲，可以将信息—物理系统理解为嵌入式系统（信息处理单元）与物理环境的组合。当希望强调物理与环境的关系时，将随时引用这一新术语。在未来，面向生物与化学的连接也会变得重要。

本书提供了信息—物理系统设计中的重要设计概念，覆盖了技术规范、硬件组件、系统软件、应用实现、仿真与验证、典型系统的优化以及测试方法。

## 嵌入式信息—物理系统的重要性

嵌入式信息—物理系统被认为是ICT未来最重要的应用领域。嵌入式系统中的处理器数量已经超过了PC，这种趋势仍将继续。据此，嵌入式软件的数量也将保持增长。人们已经预言了新的摩尔定律：消费电子产品中的代码量每两年将会翻倍[Vaandrager，1998]。嵌入式系统的重要性不断增长，它甚至影响到了美国国家研究委员会的一份报告[National Research Council，2001]。该报告中指出，“信息技术（Information Technology，IT）正在带来一场新的革命…嵌入式计算机所组成的网络…依靠对大量设备与传感器的连接，它将很有可能从根本上改变人们与环境的相互关系，它使信息以前所未有的方式进行收集、分享与处理…整个社会将因此创造一个新的里程碑”。

关于嵌入式市场规模的统计资料，可以从相关的网站上查找到。这类站点如“IT facts”[IT Facts，2010]，它分析了嵌入式系统市场的重要性。也可以从另一个角度看待嵌入式市场的规模，大部分的嵌入式处理器都是8位处理器，但除此之外，32位处理器也正被大量使用到嵌入式系统中[Stiller，2000]。早在1996年，就有人估算说每个美国人平均每天要接触超过60种微处理器[Camposano and Wolf，1996]。一些高端汽车上装置了超过100种微处理器㊀。因为人们一般并不会意识到他们正在使用微处理器，所以这个数据其实比典型的预估值还要高。关于嵌入式系统的重要性，也有如下报道[Ryan，1995]：

“工业中的嵌入式芯片驱动着我们所生存的世界的发展…它们是基于电力工作的系统的重要部分”。

根据多项预测，嵌入式系统的市场将在未来远大于PC一类的市场。美国国家科学基金会一直在资助信息—物理系统的研究工作[National Science Foundation，2010]。在欧洲，第六、第七框架项目支持嵌入式系统的研究与开发[European Commission Cordis，2010]。ARTEMIS联合组织[ARTEMIS Joint Undertaking，2010]也在政府与公司之间创建了一个合作研究组织，从而推动嵌入式计算的研究与开发。这些积极的行动，都证实了欧洲的工业界对此领域有着巨大的兴趣。当然，在其他的国家与地区也存在着类似的组织。

在当前的很多课程或培训中，都未能有效地强调嵌入式/信息—物理系统的重要性。本书的写作目的就是要改变这一现象：它将提供学习嵌入式/信息—物理系统的第一课。因此，它被设计为教科书的形式。但它又比一般的教科书提供了更多的参考，它也能帮助读者丰富关于这一领域的信息。因此，本书也适用于教师以及工程师。对于学生，本书提供的丰富的参考资料可以使其更迅速地找到相关信息。

---

㊀ 根据Personal communication的统计。

## 本书的读者

本书适合于以下4类读者：

1）计算机科学（CS）、计算机工程（CE）、电子工程（EE）专业的学生以及其他ICT相关专业对嵌入式/信息—物理系统有所研究的专业学生。本书适合对计算机软、硬件有一定基础的本科大三以上学生阅读，即适合高年级的在校生阅读。本书的目的之一，是为读者在以后的课程中探索更深层次问题铺平道路。本书假定读者具备一些计算机科学的知识，因此电子工程专业的学生可能需要去阅读一些补充材料，从而更好地理解本书的内容。事实上，本书中提及的一些内容也许已经被电子工程专业的学生所知晓。

2）一直从事嵌入式硬件但希望对嵌入式软件开发也有所涉猎的工程师。本书也将提供很多的背景资料以使读者能更好地了解相关的出版物。

3）在集中开展特定领域的研究前，希望快速而大致了解嵌入式系统技术中的关键概念的博士研究生。

4）开授关于嵌入式系统的新课程前的教师。

## 嵌入式系统课程的整合

不幸的是，在最近由ACM和IEEE计算机科学［ACM/IEEE，2008］出版的最新版计算机科学课程中，嵌入式系统并没有被覆盖到。但是，相关的应用产品的增长又形成了对此类课程的强烈需求。本书将帮助读者克服当前嵌入式系统学习中可利用的资源较少的限制。例如，对更好的规范语言、模式、工具生成、时序验证、系统软件、实时操作系统、低功耗设计技术、可靠性设计技术等的需求。本书将讲授一些问题的要点，从而作为读者展开相关研究的第一步。

## 本书的覆盖范围

本书涵盖了嵌入式系统中的软件与硬件，写作出发点可以以如下的话来解释：“嵌入式系统的开发决不能忽略潜在的硬件因素。时序、内存使用、功耗以及物理缺陷都很重要。”［Caspi et al.，2005］。

本书侧重于硬件与软件设计中的基本概念，因此除非某些工具与产品确实不同凡响，否则本书不会对其进行详细介绍。这在ARTIST中也可以解释为“如果一开始没有奠定良好的理论基础，则它们在今后的培训中将更难以学会，然而这些理论又是我们必须重视的。”［Caspi et al.，2005］。本书通过微控制器编程的讲授，来讲授嵌入式系统的设计，我们希望书中的材料不会很快过时，其中的概念在未来多年内仍将有指导意义。

关于计算机科学与工程的教科书，我在一篇论文中对此也有一些提议［Marwedel，2005］。本书的一个重要目标，即梳理出关于嵌入式系统设计中重要的相关

课题，并且讨论它们之间的联系。由此，可以避免在 ARTIST 手册中提及的一个问题：“要经历太多工业实践才能迎来一个领域的成熟，这通常又归因于文化。…课程…对于那些特别强调某项技术的课程，它们并未提供足够广泛的知识面。…因此，工厂经常难以找到训练有素的工程师”[Caspi et al.，2005]。

本书也是微控制器编程实践与理论知识的桥梁。更进一步地，它将帮助老师与学生去积极获取更多详细资料。关于某些话题，本书进行了详实的讨论，某些只作了简要概述。概述性章节的引入，是为了引入更多相关的问题与参考。这种方式使教师可以针对其所需在本书中进行选择，并查阅书中提及的参考资料。本书包含了比常规教科书更多的参考内容，这样，本书可以作为一个综合向导，为读者在进行其他资料的学习时指明方向。在实验室、项目以及独立研究中，对这些参考资料的研究也将为其带来积极作用。

关于本书的更多相关资料可以从如下网站获得：

http://ls12-www.cs.tu-dortmund.de/~marwedel/es-book

网页包含了演示 ppt、仿真工具、勘误以及其他的相关材料。发现本书错误或希望评价此书的读者可以发送邮件至：

peter.marwedel@tu-dortmund.de

在完成章节练习时，可以参考一些补充内容（如[Wolf，2001]、[Buttazzo，2002]和[Gajski et al.，2009]）。

## 预备知识

阅读本书需要对以下领域有基本认识：

1）高中水平的电气知识（如基尔霍夫定律）；

2）运算放大器（可选）；

3）计算机架构，如 J. L. Hennessy 与 D. A. Patterson 相关著作的初级水平[Hennessy and Patterson，2008]；

4）类似于门与寄存器的基本数字电路知识；

5）计算机编程（含软件工程理论）；

6）操作系统基础；

7）计算机网络基础；

8）有限状态机；

9）微控制器编程的基本经验；

10）基本的数学概念（如数组、积分、线性方程），最好能有统计学及傅里叶级数的知识；

11）算法（图形算法、优化算法以及分支界限）；

12）NP-completeness 的概念。

本书所需预备知识的架构如图 0.2 所示。

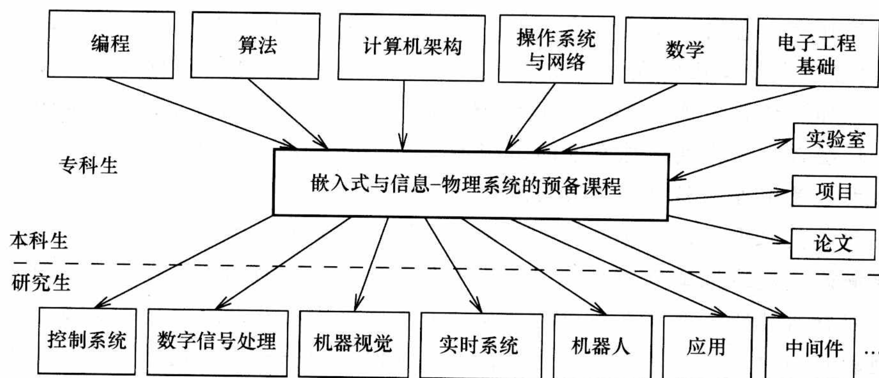



图0.2 本书知识点的架构

## 其他推荐学习资料

使用本书作教材，最好有专门的实验室，可以使用小型机器人，如Lego Mindstorms™或相似的其他机器人来做实验。可选地，也可以让学生利用有限状态机的工具来进行实际操作。

以下领域的专业课程可以作为本书的补充学习资料（参见图0.2的底行）[⊖]：

1）控制系统；

2）数字信号处理；

3）机器视觉；

4）实时系统、实时操作系统与调度；

5）中间件；

6）诸如通信、汽车、医疗设备及智能家居等应用领域的材料；

7）机器人技术；

8）传感器与执行机构；

9）嵌入式系统的特定语言；

10）特定应用的计算机辅助设备工具；

11）硬件系统的常规验证；

12）硬件与软件系统测试；

13）计算机系统的性能评估；

14）低功耗设计技术；

15）计算机系统的安全性与可靠性；

16）普适计算；

⊖ 不同学校的本科及研究生课程可能存在差异。

17）嵌入式系统的影响。

## 本书的历史

自本书第1版在2003年发行之后，嵌入式系统的快速发展带来了很多新的产物。同时，许多领域的重点也发生了改变。我们就需要根据这些变化，来重新对某些课题进行讨论。2007年本书的德语第1版出版后，我们开始了新的研究。至此，发行基本上是全新的英语版本变得很有必要，也即当前的第2版。

本书引用中未能提及版权或商标的名称，它们同样受法律保护。

欢迎您阅读本书！

Peter Marwedel

多特蒙德（德国），2010年8月

# 原书致谢

我的博士研究生，尤其是 Lars Wehmeyer，对本书的初稿进行了仔细校对。同时，参与我课程的学生也向我提供了很多有价值的帮助。David Hec、Thomas Wiederkehr、Thorsten Wilmer 和 Henning Garus 参与了本书的校对工作。以下同事及学生的意见也被整合到了本书中：R. Dömer、N. Dutt［加州大学欧文分校（UC Irvine）］、A. B. Kahng［加州大学圣地亚哥分校（UC SanDiego）］，W. Kluge、R. von Hanxleden［基尔大学（U. Kiel）］、P. Buchholz、M. Engel、H. Krumm、O. Spinczyk［多特蒙德工业大学（TU Dortmund）］、W. Müller、F. Rammig［帕德博恩大学（U. Paderborn）］、W. Rosenstiel［图宾根大学（U. Tübingen）］、L. Thiele［苏黎世联邦理工大学（ETH Zürich）］以及 R. Wilhelm［萨尔大学（Saarland University）］。在本书的写作准备过程中，还使用到了以下朋友的资料：G. C. Buttazzo、D. Gajski、R. Gupta、J. P. Hayes、H. Kopetz、R. Leupers、R. Niemann、W. Rosenstiel、H. Takada、L. Thiele 以及 R. Wilhelm。我的博士研究生小组为本书编写了章节练习。当然，作为作者，我为本书中的所有错误与不当之处负责。

我要真诚地感谢欧洲委员会对如下项目的支持：MORE、Artist2、ArtistDesign、Hipeac（2）、PREDATOR、MNEMEE 和 MADNESS，它们为本书第 2 版的写作提供了精彩的题材。

本书是使用 TeXnicCenter user interface 的 LaTeXtype 软件完成的。我也同时想感谢软件的作者为此所做的工作。

我也要感谢那些接受因作者写作本书带来了额外工作量的朋友。

最后，感谢 Springer 出版社一直推动与支持本书的出版工作。

# 目　　录

# 第1章　简　　介

## 1.1　应用领域与实例

嵌入式与 cy-phy（信息—物理）系统存在于多种领域，如下罗列了此类系统应用的关键领域：

1）**汽车电子**：在科技较为发达的国家，消费者都要求高级汽车具有较多的辅助功能，而这些辅助功能都是基于大量电子元器件实现的，如安全气囊控制系统、发动机控制系统、防抱死制动系统（Anti-Braking System，ABS）、车身电子稳定系统（Electronic Stability Programs，ESP）和其他一些安全保护功能，以及车载空调、GPS（全球定位系统）、防盗保护等。嵌入式系统同样可以减小汽车对环境的影响。

2）**航空电子**：飞机的重要价值依赖于其包含的大量信息处理设备，如飞行控制系统、防碰撞系统、导航信息系统等。嵌入式系统可以减少飞机排放物（如二氧化碳）。当然，对于飞机，可靠性是最为重要的。

3）**铁路**：铁路与汽车及飞机的情况相似，也需要极高的安全性与可靠性。

4）**电信**：移动电话是近年来增长最快的市场之一。它的射频（Radio Frequency，RF）设计、数字信号处理及低功耗设计都是其很关键的因素。当然，通信中的其他一些因素也很重要。

5）**医疗保健**：尤其是在一些老龄化的国家，医疗保健产品的重要性日益明显。将先进的信息处理技术应用到医疗设备中，这会是一个潜在的巨大市场。事实上，有多种技术都可以被应用到这一领域。

6）**安全**：随着各种安全设备市场利润的增长，嵌入式系统也可以推动这一领域的发展。如利用指纹识别传感器、人脸识别这类手段，来对访客的身份进行安全识别/认证。

如 SMARTpen®（智能笔）[IMEC，1997]，它提供了一种支付时的认证方式（见图1.1）。

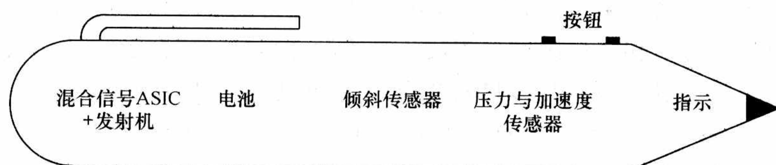



图1.1　智能笔（最初设计）

智能笔是像一支笔一样的检测仪器，它可以分析用户在签字时的一些物理特性。如倾斜、压力及加速度。这些数据被上传到主机PC，与主机所存储的用户相关数据进行对比，从而判断两者的一致性。最近，使用智能笔来记录笔体这一技术已经开始商用，当然它不仅仅局限于认证领域。

7）**消费类电子产品**：视频与音频设备是电子产品中的重要组成部分，集成到其中的信息处理单元的数量近年来一直保持增长。基于更先进的信号处理技术，这一领域提供了更多的新功能及更好的产品质量。电视机（尤其是高清电视机）、智能手机及游戏机均集成了更高性能的处理器及更大的内存，它们都是嵌入式系统在当下的典型产品。

8）**加工设备**：加工设备是嵌入式/信息—物理系统应用的传统领域，这已经有长达几十年的历史了。对此类系统，安全性比能耗更重要。图1.2（来自Kopetz的研究［Kopetz，1997］）展示了一个带排水管道的容器，管道上有一个阀门及一个传感器。根据从传感器得到的数据，计算机可以控制流向管道的液体流量。

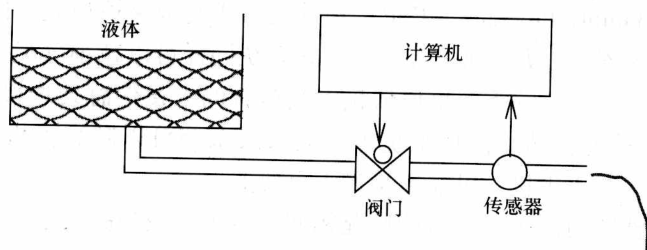



图1.2 阀门的控制

图1.2中的阀门是一种执行机构（其定义在后面会给出）。

9）**智能楼宇**：利用先进的信息处理技术，可以改善楼宇的舒适性，减少建筑内的能量消耗，增强楼宇的安全性。为了实现这一目的，楼宇内各个独立的传统子系统也必须互连。这意味着需要将空调器、照明、门禁、人群聚集及分散的信息集中到单一系统中。对于大厦内无人的房间，空调系统就可以减少运行时间与降低运行强度，其运行噪声也就会减小，灯光的照明强度也应当自动降低。智能的百叶窗系统也可以辅助空调与照明效果。建筑内可用的房间会有相应的显示，这将方便办公人员的会议安排及房间的打扫工作。在紧急情况下，仍然有人的房间可被显示在大厦的入口处（在大厦仍然有电力供应的情况下）。这样，能量将在制冷、制热及照明等环节中被节省下来，同时，大厦的安全性也得到了增强。这样的建筑最初也许只会是高档的办公楼，但这种高能效建筑的设计趋势同样会影响到私人住宅的设计。其目标之一，就是设计出**零能耗**的建筑（即建筑产生的能耗与其消耗的能耗相当）［Northeast Sustainable Energy Association，2010］。这样的设计也会为减少全球二氧化碳的排放及减慢气候变暖效应作出贡献。

10）**物流**：嵌入式/信息—物理系统可以以多种方式应用到这一领域。基于射

频识别（Radio Frequency IDentification，RFID）技术，可以在全球范围很容易地识别每个物体，并加以区分。当前的移动通信向人们提供了前所未有的接入可能性。对物流的快速性、可追踪性的要求，使嵌入式系统与物流相结合。事实上，物理—信息系统也为减少物流运作中的能量消耗作出了贡献。

11）**机器人**：机器人也是嵌入式/信息—物理系统的传统应用领域之一。机械设计对机器人非常重要，前面所描述的很多参数也将被应用于机器人设计。最近，一些可模仿动物与人类行为的机器人已经被设计出来了。图1.3就展示了这样一个机器人。

12）**军事应用**：信息处理技术应用在军事设备上已经有多年的历史了。事实上，早期的计算机就被用于分析军事雷达信号。

以上这些都展示了嵌入式/信息—物理系统的多种应用实现。在同一本书中讨论这些嵌入式系统的意义，在于它们虽然在物理构造上有很大差异，但它们的信息处理有很多相通之处。

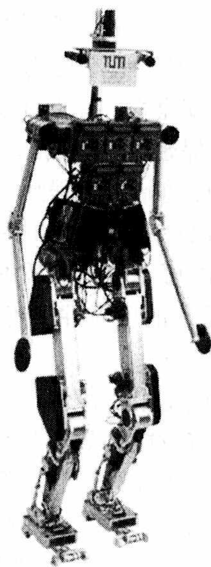


图1.3　机器人“Johnnie”（H. Ulbrich、F. Pfeiffer，Lehrstuhlfür Angewandte Mechanik，慕尼黑工业大学提供，© 慕尼黑工业大学）

## 1.2　共同特征

这些系统的共同特征如下：

1）信息—物理系统必须是**可靠的**。

很多信息—物理系统都对安全性有较高的要求，它们必须具有较高的可靠性。核电站就是对安全性有苛刻要求的系统之一，它的部分功能也是由软件控制的。当然，可靠性对其他一些系统也至关重要，如汽车、火车、飞机等。它们之所以是安全、关键的系统，重要的原因在于它们都直接与物理世界联系，并且对物理环境有直接影响。

系统的可靠因素可以用如下特性进行描述：

① **可靠性**：系统无故障地完成规定功能的能力[㊀]。

② **可维护性**：系统故障后，在规定时间内按规定方法检修，恢复到规定功能的能力。

---

㊀ 本书第5章将给出此术语的正式定义。

③ **可使用性**：系统可用于完成规定功能的程度。为了使系统具有较高的可用性，系统必须具有更高的可靠性与可维护性。

④ **安全性**：系统不会带来负面伤害或损失。

⑤ **保密性**：只有经过授权与认证，才能访问系统中的加密数据。

设计初始，工程师一般都将注意力集中在系统的功能上，想当然地认为可靠性部分可以在系统工作的后期添加。事实上，这种方式根本行不通，因为当前的设计与所需的可靠性是矛盾的。如系统的最初架构就不合理，则冗余必不可免。因此，“考虑系统的可靠性，绝不能做事后诸葛”，必须在最初设计系统时就考虑到［Kopetz，1997］。

即使是极其谨慎设计的系统，如果假定的负荷或者是可能的故障在预期之外，系统也有可能失效［Kopetz，1997］。举例来说，当一个系统在它最初假定的温度范围之外工作时，它就有可能失效。

2）嵌入式系统必须高效率。可以从如下方面来评估一个嵌入式系统的效率：

① **功耗**：功耗是评价处理器平台的重要因素。由图1.4（由H. De Man［Man，2007］提供，并基于Philips的一些信息）可以看出，这方面的技术一直在与时俱进（相应于制造工艺）。

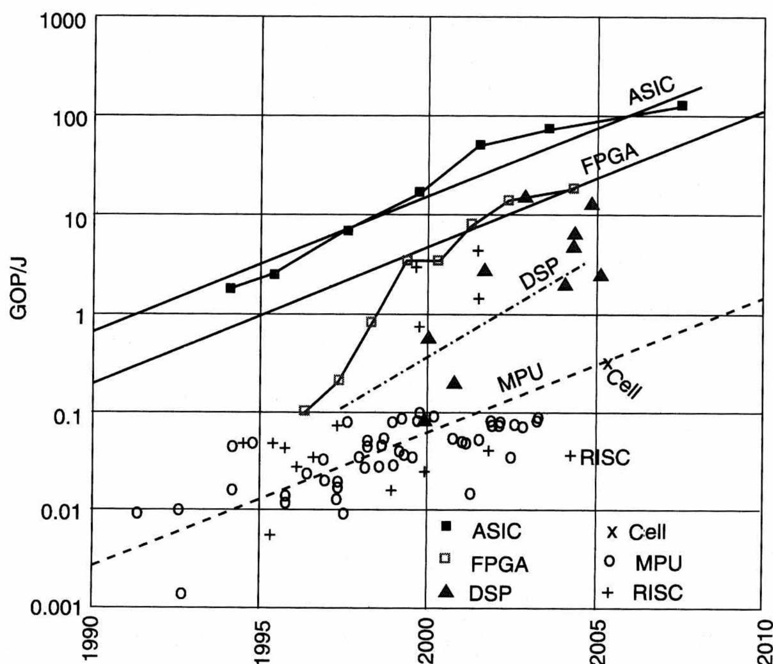



图1.4 时间和技术带来的能效改变（© Philips，Hugo de Man，2007）

很显然，随着科技的发展，集成电路的尺寸在变小，每焦耳能量所能完成的操

作[一]却在一直增多。对图 1.4 中的处理器平台，专用集成电路（Application Specific Integrated Circuits，ASIC）在单位能量中所能完成的操作最多；对于现场可编程序门阵列（Field Programmable Gate Arrays，FPGA，参见 3.3.4 节），单位能量所能完成的操作较 ASIC 要低约一个数量级。对于软件编程的处理器平台，单位能量所能完成的操作还要少一些，但这一类处理器会提供较大的软件灵活性。如 FPGA 一类的可重配器件也有一定的灵活性，但这取决于有多少功能可以采用这种灵活的设计；对 ASIC 一类的硬件设计来讲，则没有灵活性可言。灵活性与功耗的权衡对软件处理器平台同样适用：某些处理器如果针对特定的应用进行优化，如对数字信号处理器（DSP）进行特定优化，则其功耗可能也可以达到可重配器件的等级。对于通用的标准处理器，在图 1.4 中，比如 x86 一类的处理器（即 MPU）、RISC 处理器，及由 IBM 和 Sony 公司设计的 Cell 处理器，这类通用处理器单位焦耳所能完成的操作是最少的。

根据设计经验，假定一个智能手机的功耗[二]被限制在 2W 以内，这其中一半的功耗将消耗在 RF 传输、显示及音频放大上，剩下的 1W 将用于计算功能。对其进行功耗限制的原因，一方面缘于电池本身能力的限制，另一方面则是需要使设备保持在一个合理的工作温度范围内。也许在不久以后，通过技术的改进可以延长电池的工作时间，但对热量的限制仍将使人们将其功耗限制在 2W 以内。当然，对于较大的设备，应当允许其消耗更多的能量。但是出于环保方面的考虑，仍需要使设备保持低功耗。

尤其是在多媒体应用领域，对计算量的需求一直保持着高速增长，De Man 与 Philips 估计，对于高端的多媒体应用，每秒约需要 100 亿 ~ 1000 亿次的计算。图 1.4 展示的是先进的硬件技术可以在每焦耳（= Ws）能量中提供的操作数。这意味着即使是最有效的平台硬件，也不能提供我们所需的效率，这也意味着我们必须尽全力去提高它们的能效。通用处理器（如 MPU 及 RISC）基本上很难改变其低能效的特点。

这种情况也早已被预测（如根据国际半导体技术发展蓝图［ITRS，2009］的预测），能耗将是新的移动应用的关键限制。根据这份蓝图，“…这种趋势意味着，在一种适当的度量标准下，能耗的性能到 2020 年应该增长 1 ~ 2 个数量级。这就提出了如何在单位焦耳产生最大性能的问题，同时这也需要将信息理论与热力学结合起来思考”。

② **运行效率**：嵌入式系统必须尽可能地挖掘可用的硬件能力。对于特定的硬件平台，应该尽量作出最有效的应用实现。举例来说，编译器不应该引入冗余的运

---

[一] 在此处，操作指 32bit 的加法。

[二] 严格来讲，并不是消耗电力或与其相关的能量，使用电力消耗来衡量能耗的时代正在消失，更多地将使用热量来衡量系统的能耗。

行过程，因为这会导致额外的能量浪费，同时意味着可能需要使用更快的时钟频率。

③ **代码尺寸**：对于嵌入式系统，例如智能手机及机顶盒一类的系统，只有在少数情况下才能进行动态加载并运行代码。出于互连与安全性的考虑，这些场景在未来数年仍然不会太多。嵌入式系统的代码一般都直接存储在系统中。嵌入式系统一般都没有硬盘来存储程序与数据，因此希望代码尺寸尽可能小。对于所有信息处理电路都集成在单片的**片上系统**（System on a Chip，SoC），尤为如此。如果可存储代码的内存被集成在芯片内，则它需要被加以高效利用。不过，当可以采用高密度的内存（使用每卷的 bit 数来衡量）时，这一设计重点也许会发生变化。基于闪存的内存也许会对此产生较大的影响。

④ **质量**：所有便携式系统都必须是轻量级的。购买一个便携式系统时，其质量是常常被重点考虑的因素。

⑤ **成本**：对于份额巨大的大众市场，如消费类电子产品，其市场竞争已经达到白热化的状态，这就要求需要对硬件资源进行充分、高效的利用，同时也要控制软件开发的成本。对于确定的功能，应当尽量以最少的资源来实现。这意味着也要尽可能减少硬件资源与能耗，降低时钟频率与供电电压，只使用必不可少的元器件。如果不能改变最坏的运行情况，则某些组件（如过多的缓存或内存管理单元）通常可以从系统设计中省去。

3）嵌入式系统通常是与真实的物理环境相连接的，它们通过**传感器**（Sensors）收集环境信息，再通过**执行器**（Actuators）去控制环境。

**定义**：**执行器**是将数字量转化为机械动作的设备。

这种与物理环境的联系也与“信息—物理系统”相关。嵌入式系统教育一直关注于如何对微控制器编程，而往往忽略了这种关联。在这方面，信息—物理系统这一新术语，**使嵌入式系统设计从微控制器编程中得到了解放**。

4）很多信息—物理系统都必须满足**实时性**的约束。在给定时间内，如果系统的计算未能完成，则有可能导致系统信息丢失（如影响到音频或视频的质量），甚至可能给用户带来伤害（如汽车、火车或飞机未能完成预定的操作）。一些实时性的约束被称为硬约束：

**定义**：“不满足时间限制则将导致严重后果的约束，称为硬约束”［Kopetz，1997］，所有的其他时间约束都被称为**软约束**。

现在，很多信息处理系统都在使用能整体提高信息处理速度的各种技术，如使用缓存来提高系统的整体性能。在某些场合，使用重复传输某些信息来提高通信的可靠性。如以太网协议：它们在原消息丢失后，都会进行重发。一般来说，这样的重复仅仅会（希望仅是如此）导致很小的性能损失，某些信息也将产生相对于正常情况多个数量级的延时。而对于实时系统，这种性能上的折中或是延时将是不可被接受的。“**一个确定的系统响应，绝对不可以用统计学来解释**”［Kopetz，1997］。

5）典型地，嵌入式系统都是**激励系统**，定义如下：

**定义："根据物理环境决定的节奏，系统持续地与物理环境交互并等待来自物理环境的输入并进行处理，这便是一个激励系统"**［Bergé et al.，1995］。

激励系统可以视作一直处在某一状态，等待着输入信号。对于每一个输入，它们完成一些计算并产生输出及新的状态。机器人是此类系统的典型模型。但是，解决问题的算法的描述函数，却不是此类模型。

6）大部分嵌入式系统都是同时包含数字与模拟部分的**混合系统**。模拟部分在连续时间上使用连续信号，同时数字部分在离散时间上使用离散信号。

7）大部分嵌入式系统并不使用键盘、鼠标及大型的显示器作为其人机接口，它们一般使用**特定的人机接口**，如按键，滑轮、踏板等。因此，用户一般很难感觉到信息处理的过程。也正是因为如此，新时代的计算形式也被称为**消失的计算**。

8）这些应用一般也都是**面向特定应用的**，如用于控制汽车或火车的处理器，肯定不会去运行一个电脑游戏，或者去绘制表格。对此有两个主要原因：

① 运行其他程序将降低系统的可靠性；

② 只有在系统有闲置资源的前提下，才可能运行额外的程序。但对于一个高效的系统来说，是不会有这样的闲置资源的。

然而，在诸如智能手机一类的设备中，这种情况也正慢慢地发生着改变。智能手机正演化为与PC类似的系统，它已经很难再被称为信息—物理系统。据AUTOSAR（汽车开放系统架构）的介绍［AUTOSAR，2010］，这种情况也正在汽车工业中发生。

9）嵌入式系统的**教育成熟度与公众参与度均较低**。问题之一是嵌入式系统教学需要丰富的设备，这样才能激发学生的兴趣并参与进去。其次，真实的嵌入式系统都比较复杂，教学上有很大的难度。

由于这一系列的原因（最后一个除外），寻找嵌入式系统设计的通用方法，与独立分析各个不同的应用领域相比较，有着更实际的意义。

其实，并不是每一个嵌入式系统都完全具有以上特征。也可以按如下方式来定义"嵌入式系统"：**满足上述大部分特征的信息处理系统都可以被称为嵌入式系统**。这个定义有一些模糊性。然而，似乎没有必要，也无法去消除这种模糊性。

## 1.3 嵌入式系统设计的挑战

嵌入式系统都包含着大量软件。不过，嵌入式系统设计并不仅仅是软件设计单方面的工作，在软件设计的同时，还要兼顾许多其他的设计目标，具体如下：

1）嵌入式系统需要极高的可靠性，其可靠性等级要远高于传统PC系统。来看一些因为系统可靠性引发事故的例子：

① 洛杉矶机场的音量控制系统失控达3h［Broesma，2004］。问题来源于控制系统中的一台服务器，每次重启服务器后，其操作系统都有一个计数器来记录时间的长度，但这个计数器在大约48天之后就会溢出。因此，操作规范要求运维人员在每个月都重新启动一次服务器。但在出问题的那一次，运维人员刚好忘记了此事。

② 很多计算机系统故障的案例都被上报到the risks digest，它是一个面向公众的关于计算机及相关系统故障讨论的论坛（参见［Neumann，2010］）。

2）能效方面的考虑，软件设计不能脱离硬件进行想当然的开发，因此软件与硬件必须在设计流程中综合考虑。但这样非常困难，因为学术与教育机构一般都不会教育学生如何进行软、硬件结合设计。而且电子工程与计算机科学方面的结合，远没有达到我们期望的标准。基于硬件进行合理的软件应用实现，才会使系统能效最高。然而，硬件设计的高成本、长周期，意味着不能对其设计进行灵活变更，所以需要在效率与灵活性之间找到一种折中的方案。

3）嵌入式系统还必须满足许多非功能性的需要，如实时约束、能量/功耗以及可靠性等。当然，还有很多在设计中需要被考虑到的其他因素，仅仅是收集这些非功能性的需求就已经较为困难了。

4）面向物理的连接有了更多含义，如必须检查系统是否无误地满足了实时约束。对时间的管理就是众多挑战之一［Lee，2006］。

5）真实的系统都是并发的，管理并发性是嵌入式系统中的另一个主要挑战。

6）真实的嵌入式系统都是复杂的。嵌入式系统包含多种元件（Components），我们对它们的组合设计（Compositional Design）更为关注。这也意味着，我们需要了解元件组合带来的影响，如我们想知道在一辆车上加装GPS后，是否会使其通信总线的负荷过载。

7）传统的顺序编程语言并不是描述并发、定时系统的最佳方式。

图1.5中的表格强调了在进行应用实现时，对于基于类PC与嵌入式系统硬件，在设计中的主要差异。

| | 嵌入式 | 类PC |
|---|---|---|
| 架构 | 通常是紧凑架构 | 通常为非紧凑架构（x86等） |
| 与x86兼容 | 弱相关 | 强相关 |
| 固定架构? | 一般不是 | 是 |
| 计算模型（MoC） | C+多模型（数据流、离散事件、…） | 大部分为冯·诺依曼模型（C、C++、Java） |
| 优化目的 | 多种（能耗、大小、…） | 提升平均性能 |
| 实时性联系 | 非常紧密! | 非实时 |
| 应用 | 多种并发应用 | 几乎是单个应用 |
| 设计时的已知应用 | 几乎没有 | 少数（如WORD） |

图1.5 类PC硬件以及嵌入式系统硬件的应用范围

## 1.4 设计流程

嵌入式系统的设计是一项相当复杂的工作，它可以被分解为易于管理的多个子任务。部分子任务必须按顺序进行，部分子任务需要多次迭代。

设计方案最初都是从简单的想法开始，当然，这些想法必须是与实际应用相结合的。而后，这些想法成为设计文档。在设计开始之前，应该准备好标准的软件模块与硬件单元，它们应该尽可能被加以重用（见图1.6）。

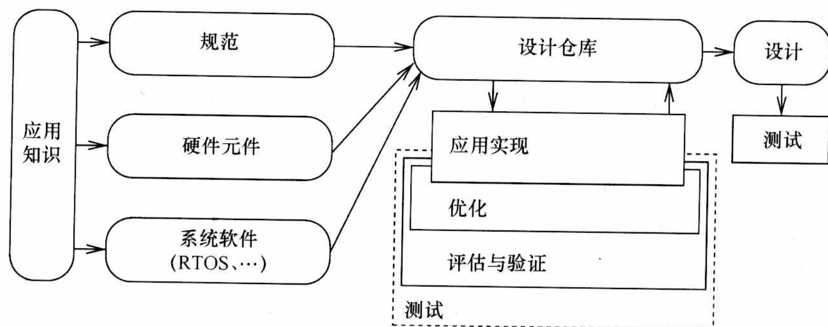



图1.6 简化的设计流程

在图1.6中（本书中的其他图与之类似），**圆角框表示数据存储，矩形框用于表示数据加工**。典型地，设计信息一般存储在**设计仓库**（Design Repository）中。设计仓库用于对设计模型进行管理。大部分情况下，设计仓库需要提供版本管理或者“修订控制”，这样的设计仓库如CVS［Cederqvist，2006］或SVN［Collins-Sussman et al.，2008］。一个优秀的设计仓库管理软件需要具备设计管理的接口，保持其对开发流程与工具的适应，这些都应该被集成到一个友好的图形化用户接口（Graphical User Interface，GUI）中。设计仓库与GUI可以更进一步扩展到**集成开发环境**（Integrated Development Environment，IDE）中，它也被称为**设计框架**（Design Framework）（如［Liebisch and Jain，1992］）。集成开发环境记录了工具与设计信息之间的依赖性。

基于设计仓库，设计方案可以迭代进行。在每一个设计步骤，设计模型都可以被检索到，然后参与决策。

在开发的迭代过程中，**应用被实现**在可执行的平台上，同时产生了新的（或者是部分的）设计信息。这种产生包含了将应用实现成并发的任务，将其功能分解到具体的硬件或软件的实现、编译、调度等步骤。

设计需要从多个方面进行**评估**，如性能、可靠性、能耗、制造工艺等。按照现有的工艺，没有哪一个设计步骤能保证绝对正确，因此有必要对设计进行**验证**。验证包括检查中间及最终的设计规范，以及其他一些资料。每一个新的设计都需要被

提前评估与验证。

源于效率对嵌入式系统的重要性，**优化**就显得非常重要。有多种可能的优化方式，如较高层次的转化方式（如使用高级循环转化），以及以节能为目标的优化。

设计迭代也包含用例准备及对可测试性的评估。如果在设计过程中，已经考虑到了可测试性的问题，则测试也应该同时在设计中进行迭代。在图 1.6 中，生成用例被作为了设计迭代中一个可选的步骤（参见其中的虚线框）。如果生成用例不在设计迭代中，则它必须在设计完成后进行。

设计仓库应该在每一个步骤完成后，进行相应的更新。

设计仓库、应用实现、评估、验证、优化、可测试性考虑及设计信息的存储，其具体的流程可能是变化的。依赖于使用的设计技术，这些行为可能会以多种形式交叉进行。

本书站在较高的层次对嵌入式系统设计进行了阐述，它并不局限于对某种特定的设计流程或工具进行讨论。因此，我们并没有给出一个详细的设计步骤。对于一些特定的设计场景，我们可以“展开”图 1.6 中的循环，并且将其与具体的设计步骤关联。例如，图 1.7 展示了基于 SpecC［Gajski et al.，2000］的一个设计流程实例。

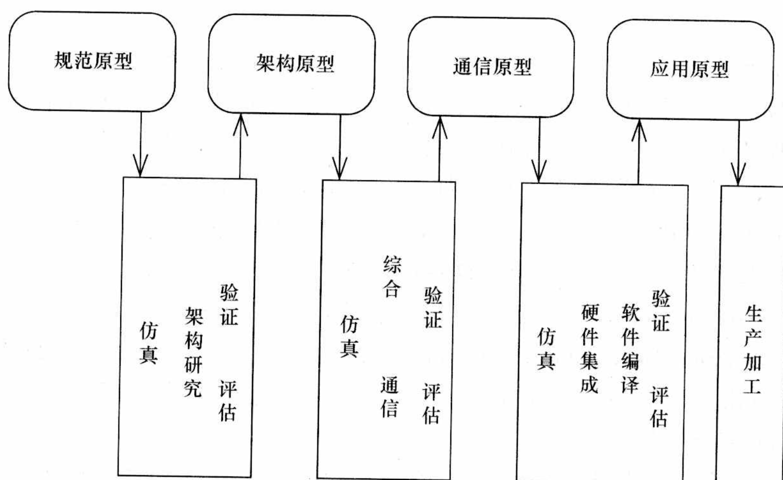



图 1.7　基于 SpecC 的设计流程实例（简化）

在这个例子中，列举了诸如架构研究、通信、综合、软件编译与硬件集成等详细步骤。关于这些术语的详细定义并不在本书的讨论范围。在图 1.7 中，每一步骤都明确地包含着验证与评估，但验证与评估并没有作为独立步骤。

图 1.8 是对图 1.6 展开的一个例子，它采用的是 V-model 的流程设计方法［V-Modell XT Authors，2010］。V-model 这种设计流程被德国很多 IT 项目采纳，

尤其是与政府业务相关的部门。

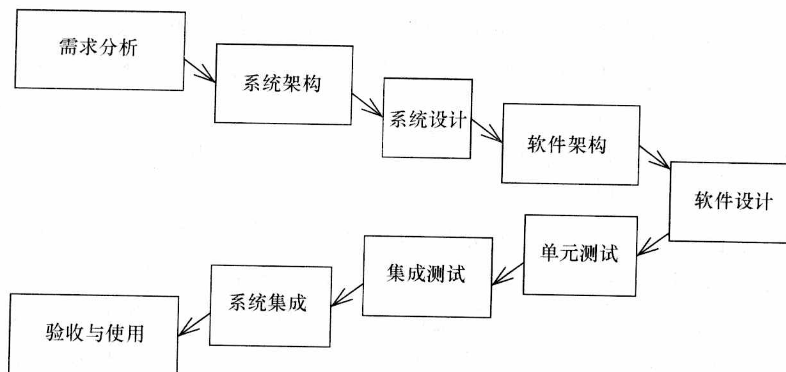



图 1.8　V-model 设计流程

图 1.8 清晰地展示了流程中的每一步需要完成的工作。这些步骤与软件开发流程（其精确定义也不在本书的讨论范围）中的确定阶段相对应。请注意，设计决策、设计评估与设计验证都在图 1.8 中被集中到一个方框内。应用知识、系统软件及系统硬件并没有在图中展开。V-model 也在图中包含了集成与测试阶段（图的下翼部分），它与图 1.6 中的测试步骤是相对应的。图 1.8 采用的是 V-model 中的“97”版绘制的。最新的 V-model XT 版本允许包含更多的设计步骤，这种更新与图 1.6 中对设计流程的解释可以更好地配合。还有其他一些迭代方法，如**瀑布模型**（**Waterfall Model**）、**螺旋模型**（**Spiral Model**），更多关于嵌入式系统软件工程的资料，可以参考 J. Cooling [Cooling，2003] 的著作。

与软件设计流程的模型一样，硬件设计也有与之对应的模型。如 Gajski 的 Y-Chart [Gajski and Kuhn，1983]（见图 1.9）就是一种非常流行的设计流程模型。

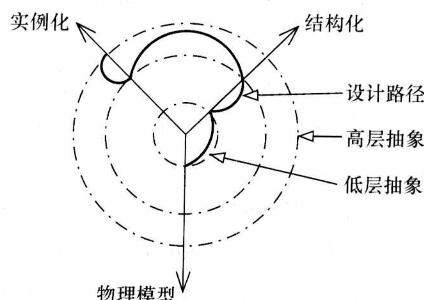



图 1.9　Gajski 的 Y-Chart 与设计路径（粗实线）

Gajski 从三个维度来考虑设计信息：行为、结构与布局。第一维仅影响行为，高层模型需要描述出设计的整体行为，分解模型则描述各个子模块的行为。第二维的模型包含结构信息，如硬件模块的信息，该维度的高层模型描述相应的处理器，低层则描述到晶体管。第三维描述了芯片的物理布局。设计路径一般从抽象的行为描述开始，到具体的几何分解描述结束。沿着这样的路径，设计流程中的每一次迭代都有步骤与之对应。图 1.9 所示的例子，首先展示的是对整体的实例化；第二步则是映射这些行为到结构化的组件，等等；最终，完成了方案实现的器件的物理布局。

前面的三张图展示了大量的设计流程，它们都使用了图1.6的迭代流程。可以讨论一下图1.6中迭代的本质。理想地，可以描述好系统的属性，然后让智能工具去完成测试。自动去产生设计细节的这一行为，这称之为综合（Synthesis）。

定义：综合是根据对期望行为的高层描述，使用与之相关的底层组件来完成系统描述的流程［Marwedel，1990］。

成功的综合将减少很多人工完成的工作。在设计系统中使用综合的目的，Gajski［Gajski et al.，1994］将其定义为“描述—综合”，这是与传统的“规范—实验—改善”，即与“设计—仿真”流程相对而言的。传统概念强调了人工设计与仿真相结合这样的事实，如怎样去发现设计中的错误。在传统设计步骤中，仿真非常重要。

## 1.5 本书的结构

与前面描述的设计流程相对应，本书的结构组织如下：第2章对特定的方法、语言与模型进行了描述；第3章描述了嵌入式系统的关键硬件组件；第4章涉及系统软件组件，尤其是嵌入式操作系统；第5章包含嵌入式系统的设计、评估与验证。将设计方案在操作平台进行实现，是嵌入式系统设计流程中的关键步骤之一。第6章包含了如何完成设计实现的一些标准方法，也包含了标准的调试技术。出于对低能耗设计的需求，这就需要更多的优化技术。于是在第7章从大量的可行优化技术中选取了部分进行阐述。第8章简要介绍了硬/软件系统集成测试。附录包含了对一种标准优化技术的描述，以及有助于更好地理解第3章中电路图的预备知识。

对于给定的应用，有必要去设计有针对性的硬件，或者优化处理器的架构。不过，本书并不涵盖硬件设计的内容。Coussy与Morawiec［Coussy and Morawiec，2008］在其著作中提供了高层硬件综合技术的总体介绍。

大部分传统的嵌入式系统图书都以大量的篇幅来描述如何使用微控制器，如处理器的内存、I/O、中断结构。有很多这样的书，如［Ball，1996］、［Heath，2000］、［Ball，1998］、［Barr，1999］、［Ganssle，2000］、［Barrett and Pack，2005］、［Ganssle，2008］、［Ganssle et al.，2008］和［Labrosse，2000］。

可以推断，随着嵌入式系统复杂度的增加，将来这类书籍的内容范围也将扩展到包含多种不同的应用示例、创建硬件模块、如何基于操作平台进行应用实现，以及评估、验证与优化技术等。本书将覆盖所有这些方面，其目的是向读者引介嵌入式系统，使其对不同应用领域都有一定了解。

如果希望获取更详实的信息，这里还推荐了一些资料（在本书的准备过程中已经用到了部分资料）：

1）有很多关于特定编程语言的资料，早期的一些资料有 Young [Young, 1982]、Burns 与 Wellings [Burns and Wellings, 1990]、Bergé [Bergé et al., 1995] 与 De Micheli [De Micheli et al., 2002]。关于新的编程语言，如 SystemC [Müller et al., 2003], SpecC [Gajski et al., 2000] 与 Java [Wellings, 2004]、[Dibble, 2008]、[Bruno and Bollella, 2009]、[Java Community Process, 2002]、[Anonymous, 2010b]，也有非常丰富的相关资料。

2）在 Kopetz [Kopetz, 1997] 的著作中，讲述了如何设计与使用实时操作系统（RTOS）。

3）在 Buttazzo [Buttazzo, 2002] 以及 Krishna 与 Shin [Krishna and Shin, 1997] 的著作中，较完整地讲述了实时调度。

4）其他一些包含嵌入式系统的著作有 Laplante [Laplante, 1997]、Vahid [Vahid, 2002]、the ARTIST road map [Bouyssounouse and Sifakis, 2005]、"Embedded Systems Handbook" [Zurawski, 2006] 以及最近由 Gajski 等人 [Gajski et al., 2009] 和 Popovici 等人 [Popovici et al., 2010] 编写的著作。

5）在嵌入式系统教育讲习会（Workshops on Embedded Systems Education, WESE）中，涵盖了嵌入式系统教育方面的内容，参见 [Jackson et al., 2009]。

6）欧洲的一些嵌入式与实时系统的网站 [Artist Consortium, 2010] 提供了大量精彩的相关内容。

7）来自于一个名为 ACM [ACM SIGBED, 2010] 的对嵌入式系统兴趣小组。

8）关于嵌入式信息—物理系统方面的研讨会文章，可以在嵌入式系统周刊（访问 www.esweek.org）及信息—物理系统周刊（访问 www.cpsweek.org）上查询。

9）机器人是与嵌入式信息—物理系统紧密相关的领域，推荐 Fu、Gonzalez 与 Lee [Fu et al., 1987] 关于此方面的著作。

## 1.6 思考题

1. 请列出"嵌入式系统"这一术语的恰当定义。
2. 怎样定义"信息—物理系统"？
3. 以一些身边可用的实例来解释嵌入式系统的重要性。
4. 比较你的前期课程与本章描述的课程，你的课程中缺少了哪些预备知识？现在有哪些高级课程？
5. 请列举出嵌入式系统的应用领域，并且至少列举 5 个嵌入式系统的例子。
6. 请列举出嵌入式系统的至少 6 个特征。
7. 从能耗方面考虑，怎样区别硬件技术上的差异？
8. 假定你的手机使用 720mAh 的锂电池，其额定工作电压是 3.7V。按恒定

1W 的功率估算，不考虑诸如电压降之类的次要因素，多长时间电池电量会消耗完？

9. 计算效率常用每瓦功率每秒所能完成的百亿次运算量有关，这与图 1.4 中描述的因素有什么区别？

10. 哪一类的实时限制被称为“硬约束”？

11. 如何定义术语“激励系统”？

# 第2章 规范与建模

## 2.1 需求

现在，将根据简化的设计流程（见图1.6），对嵌入式系统规范与建模的需求和实现进行描述。

嵌入式系统规范提供了设计中的系统模型(System Under Design, SUD)。模型可以被定义如下［Jantsch, 2004］：

**定义**："模型是实例的另一种简化，它可以是物理实体或其他形式。对于给定任务，其模型应当精确包含任务本身的特征与属性。如果它不包含除与任务相关特征以外的其他信息，则其就是关于任务的最小化模型"。

模型都使用某些语言来进行描述，这些语言都必须能描述以下特性[⊖]：

1）层次化（Hierarchy）：一般而言，人们都不太容易理解包含复杂联系的多模块（状态机、组件）系统。而描述现实中的系统，就需要很多不同的模块，这会使人们更难以理解。层次化（与抽象相结合）是解决这种困境的关键机制。层次化使人们在很多时候都只需要处理很少量的系统模块。

有两种层次化的方法：

① 行为级（Behavioral Hierarchies）：它的各层模块包含了描述系统行为的必要信息，这类模块如状态、事件以及输出信息。

② 结构级（Structural Hierarchies）：它描述物理组件如何组成了整个系统。

举例如，嵌入式系统由处理器、内存、执行器、传感器等器件组成。处理器又包含着寄存器、多路复用器以及加法器，多路复用器又是由多个门单元组成的。

2）基于模块化的设计（Component-based Design）［Sifakis, 2008］：从一个系统所含的各个模块的功能，可以"容易地"推断出整个系统的功能。如果将两个模块连接在一起，则新的功能一般也是可以猜测的。

举例如，假定在一辆汽车上新增了模块（如GPS单元），则其产生的额外功能（如线路等）应当是可以被预测的。

3）并发性（Concurrency）：实际系统都是分布式、由多个模块组成的并发系

---

⊖ 此处参考了Burns等人［Burns and Wellings, 1990］、Bergé等人［Bergé et al., 1995］与Gajski等人［Gajski et al., 1994］的相关资料。

统。因此，也很有必要规范系统的并发机制。不幸的是，由于对并发系统的可能行为缺乏认识，人们并不擅长理解并发系统以及实际系统中的相关问题。

4）同步与通信（Synchronization and Communication）：各个模块都应该可以进行通信与同步。如果没有通信，组件之间就不能协同工作，也就只能独立地使用单个组件。组件也需要协商如何对共有资源进行使用，如协商如何进行必要的互斥操作。

5）时序行为（Timing-behavior）：很多嵌入式系统都是实时系统。清晰的时序是嵌入式系统的特征之一。事实上，对于“信息—物理系统”，时序模型更重要。时序是实际系统的一个重要组成要素，因此必须根据特定的嵌入式/信息—物理系统，去挖掘其相应的时序需求。

然而，计算机科学方面的标准理论只能以非常抽象的形式对时序进行建模。O-notation 就是这方面的例子。标记仅仅反映了功能的增长速率。它常用于对算法的运行时间进行建模，但并不擅长描述系统的真实执行时间。对于确实的物理量，都有计量单位对其进行描述，但 O-notation 没有，因此，它并不能区分一个世纪与 1s。关于算法的终止行为，也有一些差异：标准理论要求算法能最终结束，但对实时系统，需要在给定时间结束算法。

E. Lee 对此种情况导致的问题作了一个清晰的总结：“从嵌入式软件的观点出发，核心抽象（在计算机科学上）缺乏对时序的支持，这是一个缺陷”［Lee，2005］。

根据 Burns 与 Wellings［Burns and Wellings，1990］的理论，时序建模需要从以下 4 个方面进行考虑：

① 测量消耗的时间：对于很多应用，有必要去统计从计算开始到当前，已经消耗了多少时间。通过读取一个计时器，可以做到这一点。

② 处理器延时特定时间的意义：实时编程语言一般都有产生延时操作指令。但不幸的是，这类操作指令应用在嵌入式系统软件中，并不能产生精确的延时。假定 $T$ 任务需要被延时 $\delta$[⊖]，这时，操作系统一般会将 $T$ 任务的状态从“就绪（Ready）”或“运行（Run）”，转为“挂起（Suspended）”。在经历 $\delta$ 的延时后，$T$ 的状态再从“挂起”变为“就绪”。这并不意味着任务此时就真正开始执行。如果一些高优先级的任务正在运行，或者操作系统是非抢占式的，则 $T$ 任务的延时将大于 $\delta$。

③ 可以指定超时：很多情况下，需要去等待一系列事件的发生。然而，这个事件可能并不能在给定的时间间隔中发生，人们也希望及时知道这一点。如可能正在等待来自网络连接的响应，如果在时间 $\delta$ 内仍然没有收到响应，也希望得到通知，这就是超时（Timeouts）的目的。实时编程语言一般都会提供产生超时的方

⊖ 在本书中，不区分进程、线程与任务。

法，但使用超时机制常常也会遇到跟处理器延时一样的问题。

④ 指定死限与调度的方法：某些应用场景要求计算与处理都必须在限定的时间内完成。如如果汽车上的传感器检测到事故发生，则需要在 10ms 内弹出安全气囊。在此背景下，必须保证软件能在 10ms 内作出是否弹出安全气囊决定。如果气囊弹出过晚，很可能会危及到乘客安全。大部分的编程语言并不支持时序约束，如果在以后这种情况有所改变，则可以通过单独控制文件、弹出菜单等来指定时序。即使是这样，情况仍然不容乐观：很多流行的硬件平台，它们的时序受多种因素影响，即缓存、流水线暂停、投机执行、任务抢占、中断等，它们都使硬件的执行时间难以预测。同样地，**时序分析**（Timing Analysis）（对时序约束的验证）也是一个非常困难的任务。

6）**状态机行为**（State-oriented Behavior）：在第 1 章已经提及，机器人在激励系统建模时提供了很好的机制。因此，使用机器人系统可以比较容易地描述基于状态机的行为。但是，传统的机器人模型有很多不足，因此它们不能进行时序建模，并且也不支持层次化的抽象。

7）**事件处理**（Event-handling）：由于嵌入式系统是典型的激励系统，需要有可以对事件进行描述的机制。这些事件可以来自系统外部（由环境触发），也可以来自系统内部（来自 SUD 的模块）。

8）**异常行为**（Exception-oriented behavior）：在很多实际系统中，均可能会发生异常。为了设计一个可靠的系统，就必须较容易地描述异常行为的处理。在系统的每个状态（如在传统的状态图中）都设计异常处理机制，这显示是不可行的。如在图 2.1 中，输入 $k$ 有可能会导致系统异常。

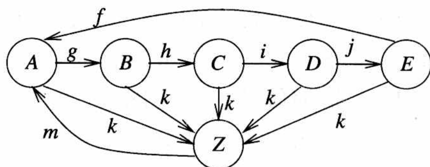


图 2.1　包含异常 $k$ 的状态图

在每个状态都指出此异常，将会使图 2.1 变得非常复杂。对于一些有多种转换的大型状态图，这种情况会更糟糕。下面将描述如何将所有转换集中到一起。

9）**编程元素**（Presence of Programming Elements）：对于表达指定的计算，流行的编程语言已经被证明是很方便的方式。因此，编程语言的元素都可以在使用的技术文档规范中查找到，传统的状态图并不能满足这一需求。

10）**可执行性**（Executability）：规范并不能自动与人们头脑中的想法同步。执行，则是对这些想法可行性的检查。此时，使用编程语言的规范就有很显著的优势。

11）**可支持大型系统设计**：嵌入式软件项目正向着大型化、复杂化的方向发展。软件技术中已经有设计一些大型系统的机制，如单元化就是其中之一，这在规范方法学中可以学习到。

12）**支持特定领域**（Domain-specific Support）：如果同样的规范技术可以被应用到所有不同类型的嵌入式系统上，那将是非常美好的事情。因为这可以减少开发

规范技术的工作量，也减少了很多支持工具。但由于应用领域的多样化，不太可能做到某一种语言可以被高效地应用到所有领域。如控制领域、数据领域、集中式或分布式应用领域，它们都可以从针对该领域的编程语言中相应受益。

13）**可读性**（Readability）：人们必须要能阅读规范，否则也就无法验证它是否满足了 SUD 中指明的目标。所有的文档也应该可以被计算机识别。因此，人与计算机都需要能识别以某种语言描述的规范。

初始阶段，这些规范可以采用英语或日语等自然语言进行描述，然后形成设计文档，最终的实现也可以与这些原始文档进行对比。但在后续的设计阶段，自然语言就不再适用，因为它缺乏规范技术的一些关键因素：它需要能够去检查规范的完整性与缺失，同时也能够系统地从规范中得到实现方法。可见，自然语言并不满足这些需求。

14）**便携性与灵活性**（Portability and Flexibility）：规范最好是独立于特定的硬件平台，这样它们才能较容易地在多个目标平台上使用。理想的情况是，硬件平台的改变对规范没有影响。在实际中，一些小的变更应该是可以被接受的。

15）**终止**（Termination）：应当允许从规范中结束一个进程。这意味着在我们使用的规范中，挂起故障任务（断定一个算法是否应该终止，参见［Sipser，2006］）是完全可行的。

16）**支持非标准的 I/O 设备**：许多嵌入式系统使用的 I/O 设备，都与 PC 上的典型设备不一样，应当可以较为方便地描述这些系统的输入与输入。

17）**附加属性**：真实的 SUD 同时需要有许多非功能性的特征，如容错、尺寸、扩展性、预期寿命、能耗、质量、可回收性、用户友好性、电磁兼容（EMC）性等。在一个正常设计的系统中，全部考虑到这些特征是非常困难的。

18）**对设计可靠系统的支持**：规范技术应该提供对设计可靠系统的支持。如规范语言应该语义清晰，促进正式的验证，有能力去描述安全性及安全需求。

19）**便捷地生成高效应用**：出于对嵌入式系统高效性的要求，规范也应该能帮助系统生成高效的实现。

20）**适当的计算模型**（Model of Computation，MoC）：与通信技术相结合的顺序执行的冯·诺依曼模型是一种常用的 MoC。但是，这样的模型有一系列的问题，尤其是在嵌入式系统应用中，具体如下：

① 缺乏时序描述能力。

② 冯·诺依曼模型一般基于全局共享内存（如在 Java 中）进行访问。它必须保证对共享资源的互斥操作，否则在允许任务抢占的多线程系统中，将可能引发一些不可预见的问题[⊖]。使用原子操作是进行资源互斥的常见方式，但它可能会引起

⊖ 在操作系统的课程中一般会有这样的例子。

死锁。在系统中，死锁非常难以检测，可能会存在数年之久。

Lee［Lee，2006］在此方面提供了一个很有提示意义的案例，他研究了 Java 中简单的观察者模式的应用。在这个模式中，消息源必须将数值的改变通知给所有登记过的观察者。在嵌入式系统中，这是一种非常常见的模式，但在一个有抢占的冯·诺依曼环境中，这种模式却很难实现。Lee 给出了基于多线程的 Java 运行环境的可能实现代码：

```
public synchronized void addListener(listener) {...}
public synchronized void setValue(newvalue) {
 myvalue=newvalue;
 for (int i=0; i<mylisteners.length; i++) {
  myListeners[i].valueChanged(newvalue)
 }
```

addListener 方法用于新的观察者登记，setValue 方法用于将新数值通知给已经登记过的观察者。简言之，在一个多线程环境中，线程是可以被随时抢占的，这将导致当前线程的执行被强制打断。当 setValue 正在执行时，如果此时登记新的观察者，则可能因程序的并发性出现问题。如无法知道新的数值是否也能被新登记的观察者接收到。另外，所有的观察者组成了此类的一个全局数据结构。因此，为了避免部分数据已经发送变更通知，同时观察者集合又有变化的情况，这些方法必须进行同步。这样的话，在某个给定的时间内，只能是两种方法之一处于运行状态。在多线程环境中，为了阻止对当前执行方法的非期望打断，互斥操作是很有必要的。为什么上述代码可能会有问题呢？因为 valueChanged 方法可以对某些资源（如 R）进行独占式的访问，如果资源是为某些其他方法而分配的（如 A），那么只有在 A 释放了资源 R 后，valueChanged 才能访问 R。如果 A 在释放资源 R 之前，还调用了（可能是间接的）addListener 或 setValue，那么将产生死锁：setValue 在等待 R，而 R 需要 A 去释放，但 A 又只能在 addListener 或 setValue 返回之后才能释放 R，这样就产生了死锁。

这个例子展示了在被抢占的多线程条件下产生了死锁，因此多线程需要对关键资源进行互斥访问。Lee［Lee，2006］认为，给此类问题的诸多建议“解决”方案，它们本身就是存在问题的。所以，在多线程的冯·诺依曼环境中，如此简单的模式也非常难以正确实现。这个例子也展示了人们很难理解并发性，即使是经过严格的代码检查，它们仍然是较大的潜在风险。

Lee 总结认为“使用线程、信号量与互斥锁编写的非常规软件，使人非常费解”，以及“嵌入式系统与并发模式并不兼容。…嵌入式系统正常工作…仅需要高效的调度策略就足够了”［Lee，2005］。

关于死锁产生的根本原因，在关于操作系统的书中有详细研究（参见［Stallings，2009］）。从这些书中可以总结出 4 个常见的可能死锁的场景：互斥操作、对

资源没有抢占、在拥有资源的同时又等待资源、线程之间的循环依赖。很显然，这4个场景都出现在了上面的例子中。操作系统理论并不会提供解决此类问题的通用方法。PC系统有时可以接受一些概率极低的死锁，但安全设备绝不能接受任何死锁。

我们希望通过指定SUD从而不再过多地去关心死锁问题。因此，有必要研究非冯·诺依曼的MoC，从而避免这一问题。从2.2节起，将研究这样的MoC。读者可以看到，在这样的MoC中，可以很容易地实现观察者模式。

从上述列举的诸多需求可知，已经很难有一种单一的常规语言有能力完全满足这些需求。因此在实践中，必须去采取一些折中的方案，同时可能需要使用多种语言（每一种语言应该适合描述某一类问题）。对于一个实际的设计问题，如何选择语言，应该根据应用的领域，以及产品工作的环境来决定。接下来，将对可应用于实际设计的多种语言进行总体介绍，这些语言将展示相应计算模型的本质特性。

## 2.2 计算模型

计算模型（MoC）描述了对于计算的一种设想机制。在常见的案例中，系统都包含了诸多模块。在实践中，一般严格区分模块中的计算与通信中的计算。于是，MoC可以定义（参见［Lee，1999］，［Janka，2002］，［Jantsch，2004］，［Jantsch，2006］）如下：

1）**模块**（**Components**）与其中的计算分布：程序、进程、函数、有限状态机都是可能的模块。

2）**通信协议**：通信协议描述了模块之间通信的方式。基于异步的消息传输与重组的通信是通信协议中的典型例子。

可以以一个简图来描述模块之间的关系。简图中提及的计算也可理解为进程或任务，相应地，简图中的关系也可以理解为**任务图**或**进程网络**。简图中的节点代表了执行计算的模块。输入数据流经过计算后成为输出数据流。通常，计算都是使用一些较高层的编程语言实现的。迭代就是一种典型的计算（有可能是非终止的迭代）。在每一个迭代的周期中，输入数据被处理并加工，然后产生相应的输出数据流。连线表示了模块之间的关系。现在将要详细对简图进行介绍。

计算模块之间最明显的关系即它们的顺序依赖：只有在某些计算完成后，另一些计算才能开始执行，这种依赖一般以**依赖图**（**Dependence Graph**）进行描述。图2.2就展示了一组计算之间的依赖图。

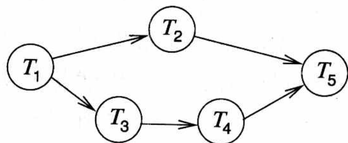



图2.2　依赖图

**定义**：依赖图是一个有向图$G=(V, E)$，其中$V$是节点的集合，$E$是边的集合。

边 $E \subseteq V \times V$ 决定了节点 $V$ 之间的关系。如果 $(v_1, v_2) \in E$，则 $v_1$ 称为 $v_2$ 的**直接前驱**（Immediate Predecessor），$v_2$ 称为 $v_1$ 的**直接后继**（Immediate Successor）。假定 $E^*$ 是 $E$ 的传递闭包（Transitive Closure）。如果 $(v_1, v_2) \in E^*$，则 $v_1$ 是 $v_2$ 的**前驱**（Predecessor），$v_2$ 是 $v_1$ 的**后继**（Successor）。

这样的依赖图形成了任务图的一个特例。任务图一般会包含比图2.2更多的信息，如它们会包含如下扩展信息：

① **时序信息**：任务可能有启动时间、死限（Deadlines）、周期与执行时间。为了在计算的调度中也将这些因素考虑进去，那么将这些信息也包含在依赖图中就非常有用了。使用Liu［Liu，2000］所著书中的方法，在图2.3中包含了可能的执行时间间隔。对 $T_1$ 和 $T_3$ 的计算是独立进行的，括号中的第一个数字表示了任务的启动时间，第二个数字表示死限（图2.3中没有标明执行时间）。如 $T_1$ 应该在时刻0开始，同时其应该最晚在时刻7结束。

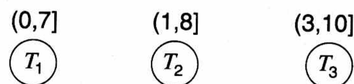


图2.3　包含时序信息的图

事实上，更多复杂的计算时序与依赖都可以引入到图2.3中。

② **计算模块之间不同关系的区别**：优先级关系是对可能的执行顺序约束的建模。更进一步讲，它对于区分计算的调度约束与通信约束非常有用。通信可以用边来描述，但对于每一个边，也许还会有补充的信息，如通信时间、信息量等。描述优先级的边可以与其他的边不一样，因为对于有些即使没有信息交换的模块，计算也必须按顺序执行。

在图2.2中，并没有明确描述输入与输出（I/O）。这意味着可以假定图中的计算没有优先级关系，计算模块可能会不定时收到来自输入的数据。同时，产生的输出将被用于子模块的输入，只有计算结束后，这些输出才是可用的。对于输入与输出，通常都需要作出明确的描述。为了实现这一目标，就需要另外一种关系。使用另外一种符号［Thoen and Catthoor，2000］，它以部分填充的圆环来表示输入与输出。在图2.4中，这种圆环表示了I/O的边。

③ **对资源的互斥访问**：对于某些资源，要求各个计算模块对其访问是互斥的，如对于一些I/O设备，或者用于通信的一些内存区域。在计算调度中，必须考虑到哪些操作需要互斥。这些信息对于系统设计非常有用，如它们可以用于避免优先级反转的问题。在图2.4中，同样也包含了对资源互斥操作的一些信息。

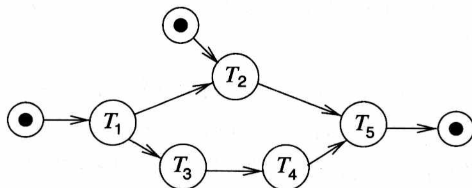


图2.4　包含I/O节点与边的图

④ **周期性调度**：许多计算过程，尤其是数字信号处理，都是周期性的。这意味着必须很仔细地区分任务与它的执行［后者常被称为作业（Job），［Liu，

2000]]。这样的任务调度图都是无限循环的。图2.5展示了一个任务图，它包含$J_{n-1}$到$J_{n+1}$这样的多个作业。

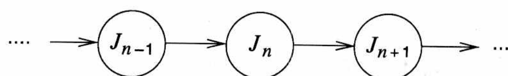



图2.5 含有多个作业的图

⑤ 节点层次图：在图中以节点的形式来描述计算的复杂度，这与前面的几种方式会有一些差异。一方面，特定的计算可能会非常复杂，它可能包含上千行的程序代码。另一方面，程序可以被分成很多个小的程序片段，每个节点只与一个操作相对应。节点图的复杂度被称为粒度（Granularity）。应该选择哪一种粒度呢？对此并没有一个通用的答案。某些情况下，粒度可能要越大越好，如将每个节点等同地视为进程，它们被实时操作系统（RealTime Operation System，RTOS）调度，这时就应该采用大的粒度，从而使不同进程之间的上下文切换最小化。而对于某些情况，可能需要以一个节点来表示单一的操作，如在节点需要与硬件或软件有对应关系时。如果一系列操作［如频繁地使用DCT（Discrete Cosine Transform，离散余弦变换）］可以被映射到特定功能的硬件，则它不应该使用包含很多其他复杂操作的节点来表示，而是应该以适当的模型独立表示。为了避免对粒度频繁改动，节点层次图就非常有用了。在较高层，节点可以表示复杂的任务，较低层可以表示一些基本模块[⊖]，更低些的层可以表示一些计算操作。图2.6展示了图2.2中的层次图，它使用矩形来表示一个层次节点。

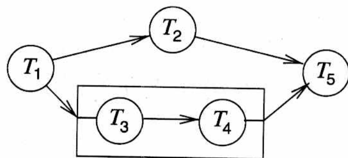



图2.6 层次任务图

如上所述，MoC可以根据通信模型（以任务图之间的边来反映）与模块之间的计算模型（以任务图中的节点来反映）进行分类。下面，将介绍这些模型中的典型例子：

3）通信模型：下面将通过讨论来区分两种通信模型：**共享内存**（Shared Memory）与**消息传递**（Message Passing）。对于其他一些通信模型（原子通信［Bouwmeester et al.，2000］），不在本书的讨论范围中。

① **共享内存**：对于共享内存，所有模块都去访问相同的内存区域，从而完成通信。

除非对某个共享内存区域只有读操作，否则必须考虑多个模块对其访问时的互斥。对于必须保证互斥操作的程序代码，也称为**临界段**（Critical Section）。有很多种被建议的方式，都可以保证对资源的互斥访问：如信号量、条件临界段、条件监视等。可以查阅关于操作系统的书（如Stallings的相关著作［Stallings，2009］）来了解这些方式的技术细节。共享内存的通信效率比较高，但对于在硬件上没有共享

---

⊖ 基本模块可以理解为在代码的每个分支进行同类操作。

内存的多核处理器，这一方式的实现就非常困难了。

② **消息传递**：消息传递，包含了消息的发送与接收过程。即使处理器没有共享内存，仍然可以实现消息传递。但是需要注意的是，消息传递的效率比共享内存要低不少。对于此类通信，需要区分以下 3 种技术：

a. **异步消息传递**（Asynchronous Message Passing），或称为**非阻塞通信**（Non-blocking Communication）：在异步消息传递模式下，模块通过向能缓存消息的通道（Channels）发送消息，发送方不需要知道接收方是否准备好接收消息。在现实中，这就相当于发送一封信或一封电子邮件。这种通信方式也有潜在的问题：发送的消息需要被缓存，而消息缓冲区可能会溢出。这种通信调度方式还有很多个版本，如通信有限状态机与数据流模型。

b. **同步消息传递**（Synchronous Message Passing），或称为**阻塞通信**（Blocking Communication）、**基于会合的通信**（Rendezvous Based Communication）：在阻塞通信模式中，模块基于原子操作进行通信，这种即时行为被称为**会合**（Rendezvous）。通信的一方，必须等待相应的另一方也同时准备好。这种通信模式，在现实生活中就像是会面或是打电话一样。它没有缓冲区溢出的风险，但通信效率会受到影响。基于这种模式的语言如 CSP 和 ADA。

c. **扩展的会合模式、远程调用**（Extended Rendezvous，Remote Invocation）：在这种模式下，发送方只有在收到来自接收方的应答后，才能继续执行后续过程。在收到消息后，接收方不一定需要马上产生应答，可以在对消息进行一些初步检查后，再发送应答信号。

4）模块间计算的组织：

① **冯·诺依曼模型**（von-Neumann Model）：这种模型基于对有序的独立计算的顺序执行。

② **离散事件模型**（Discrete Event Model）：在这种模型中，事件都携带着有序的时间戳，从而表明了事件的发生时间。离散事件仿真器一般都有一个全局的事件队列，按顺序存储着多个事件，这些事件也会按存储顺序被处理。这种模型的缺点，就在于它依赖于来自事件队列的全局通知序列，很难将其转化为并行语言的实现。如 VHDL、SystemC 以及 Verilog。

③ **有限状态机**（Finite State Machine，FSM）：这种模型依赖于有限状态机的通知、输入、输出，以及状态机的转变。多个状态机之间进行的通信，称为**通信有限状态机**（Communication Finite State Machine，CFSM）。

④ **微分方程**（Differential Equation）：微分方程很适合对模拟电路以及物理系统进行建模。它们也可以用于对信息—物理系统建模。

5）**组合模型**（Combined Model）：实际语言一般都结合了多种模型，这些模型内的多个模块之间又存在着不同的通信组织。如 StateCharts 组合了共享内存的有限状态机；SDL 结合了异步通信有限状态机；ADA 与 CSP 结合了同步消息传递的

冯·诺依曼模型。图2.7列出了在本章将继续讨论的组合模型，同时也列出了针对大部分MoC的语言。

| 模块的通信组织 | 共享内存 | 消息传递<br>同步 | 消息传递<br>异步 |
|---|---|---|---|
| 未定义的模块 | 纯文本或图，用例 | （消息）时序图 | |
| 通信有限状态机（CFSM） | StateCharts | | SDL |
| 数据流 | （未使用） | | Kahn 网络<br>SDF |
| Petri nets | | C/E nets、P/T nets、… | |
| 离散事件（DE）模型⊖ | VHDL、Verilog、SystemC | （仅在成熟系统中）<br>在Ptolemy中的分布式DE | |
| 冯·诺依曼模型 | C、C++、Java | C、C++、Java、…库<br>CSP、ADA | |

图2.7　MoC概述与备选语言

⊖ 对于VHDL、Verilog与SystemC的分类，基于这些语言在模拟器中的应用情况。在模拟器内核的“顶层”，可以对消息传递进行建模。

不同的MoC有其不同的擅长应用领域，针对某一应用领域，选择“最好”的MoC往往比较困难。使用混合MoC（如［Davis et al.，2001］描述的Ptolemy框架）也许是一种比较好的选择。同时，模型也许需要从一种MoC转变为另一种，非冯·诺依曼模型常常转变为冯·诺依曼模型。如果从两种模型之间相互转化比较容易，那么也就意味着可以较少地关注模型之间的差异。

从非冯·诺依曼模型开始的设计常被称为**基于模型的设计**（**Model-based Design**）。基于模型的设计的关键想法，即得到设计中的系统（System Under Design，SUD）的一些抽象模型，然后可以基于这些模型来研究系统的各种特性，但并不关心软件代码。在模型被逐步细化明晰之后，软件代码就随之很自然地得到了实现。关于“基于模型的设计”并没有一个精确的定义，它常与控制系统相关，包含一些传统的控制组件，如微分器、积分器等。但这种观点太过严格，因为同样可以从消费系统进行抽象。

接下来，将对不同的MoC进行讲述，使用现有语言来展示它们的特性。Edwards［Edwards，2006］提供了一份相关的调查报告（但比较简短），同时更多的资料可以参见［Gomez and Fernandes，2010］。

## 2.3　早期设计阶段

关于系统的最初想法一般都以非正式的方式进行记录，如在几张简单的纸上。通常，在项目设计的早期阶段，都是以英语、日语这类自然语言来对SUD进行描

述，一般都是日常使用的文件格式。这些描述最终都应该以可使用计算机处理的格式记录下来，它们需要用一些文字处理软件进行加工，并且使用工具来管理这些文档。优秀的管理工具在进行版本管理的同时，也能维护需求与依赖分析等之间的链接关系。

DOORS® ［IBM，2010b］就是这样一种工具。

### 2.3.1 用例

对于一些应用，针对SUD预先设计一些可能的使用场景会非常有用。这样的一些使用场景，称之为**用例**（Use Case），用例描述了SUD的可能应用。不同的用例应当使用不同的标记。

UML（Unified Modeling Language，统一建模语言）标准［Object Management Group（OMG），2010b］、［Fowler and Scott，1998］、［Haugen and Moller-Pedersen，2006］常常被用于系统早期的规范设计阶段。UML由一群软件技术专家设计，并且由商业工具支持。UML的主要目标是对软件设计流程进行支持，它也针对用例提供了标准化的框架。

从设计者的角度，很难对用例进行精确计算或者通信建模。一方面来自于建模本身的难度，另一方面却有可能是故意为之，其目的是为了避免在设计早期过多关注细节，这也是常常存在争议的地方。

如图2.8展示了为一个电话应答机设计的用例[1]：

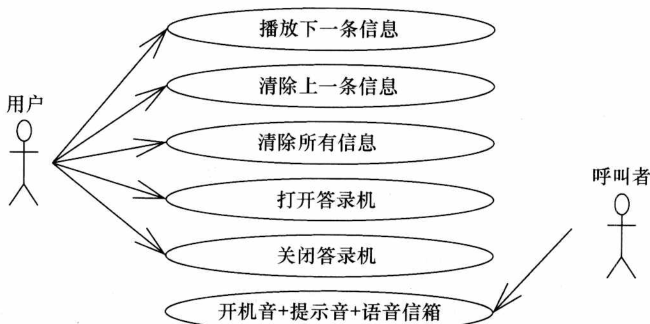



图2.8 用例的例子

基于SUD对应用的支持，用例将不同的用户进行分类，从而可以在较高的抽象层次上收集使用者的期望。

[1] 假定UML已经在软件工程学的课程中进行了深入介绍，因此在本书中不作过多讨论。

### 2.3.2　（消息）序列图

在更细化的层次上，为了应用一些 SUD 的用例，希望明确地标识出模块之间必要的通信消息序列。**序列图**（Sequence Charts，SC），其在早期也被称为**消息序列图**（Message Sequence Charts，MSC），它就提供了这样的机制。序列图使用二维图中的一维（一般是竖轴）来标识序列，第二维表示不同的通信模块。SC 描述了消息传输的部分时序，同时也展示了 SUD 的可能行为。

SC 是 UML 标准化的模块之一。相对于 UML1.0，UML2.0 扩展了 SC 的功能，允许对组件进行更加详细的描述。图 2.9 展示了一个电话应答机的例子。

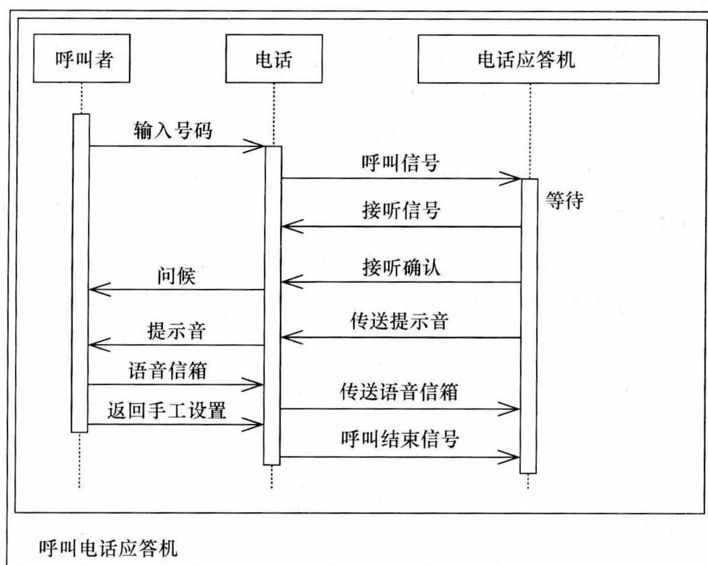



图 2.9　电话应答机的 UML 模型

虚线被称为“时间线”，消息在时间线上是有序排列的。在本例中，假定所有的信息都是以消息的形式传送的。图中的箭头表示了异步传送的消息，即在这类消息的传送中，发送方不必等待来自接收方的确认，它可以发送多条消息。本例中的应答机根据设定的时间等待用户接听电话，如果超过设定时间后仍然无人接听，则应答机将自动接听并向呼叫方发送自动应答提示，此时呼叫者可以留言。其他一些时序（如呼叫者提前结束通话，或者被叫者接听了电话）并未在此处列出。

SC 并不适合描述基于控制的复杂行为，这时需要使用一些其他的 MoC。通常需要适合一些先决条件时才能采用 SC，这种先决条件，可能发生的时序与必然发生的时序之间的区别，以及其他一些扩展场景，被称为真实序列图（Live Sequence Charts）[Damm and Harel，2001]。

**时间/距离图**（Time/Distance Diagrams，TDD）是 SC 一种常见变化形式，在

TDD 中，竖轴不仅表示序列，还表示了真实时间。在某些情况下，横轴同时代表着模块之间的真实距离。

TDD 为列车或公共汽车的调度提供了最准确的虚拟模型。图 2.10 就是这样的实例。

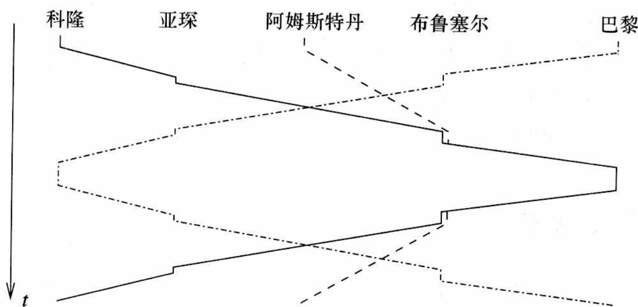



图 2.10 TDD

图 2.10 中的例子展示了在阿姆斯特丹、科隆、布鲁塞尔以及巴黎之间的列车调度过程。列车可以从阿姆斯特丹或者科隆出发，经过布鲁塞尔到达巴黎，亚琛是科隆与布鲁塞尔之间的一个中转站。竖轴相应地表示了各站点之间经历的时间。从图中可知，来自科隆与阿姆斯特丹的列车会同时到达布鲁塞尔，巴黎与科隆之间还有一趟经过阿姆斯特丹的列车，但它不属于阿姆斯特丹车站。

本例以及其他一些类似例子，都可以使用 levi 仿真软件来进行仿真 [Sirocic and Marwedel，2007d]。图 2.11 展示了规模更大、更加真实的例子，它 [Huerlimann，2003] 描述了瑞士洛书堡（Lötschberg）地区的铁路交通情况。快车与慢车在图中以它们的斜率来进行区分，这种建模的技术在现实中非常常见。

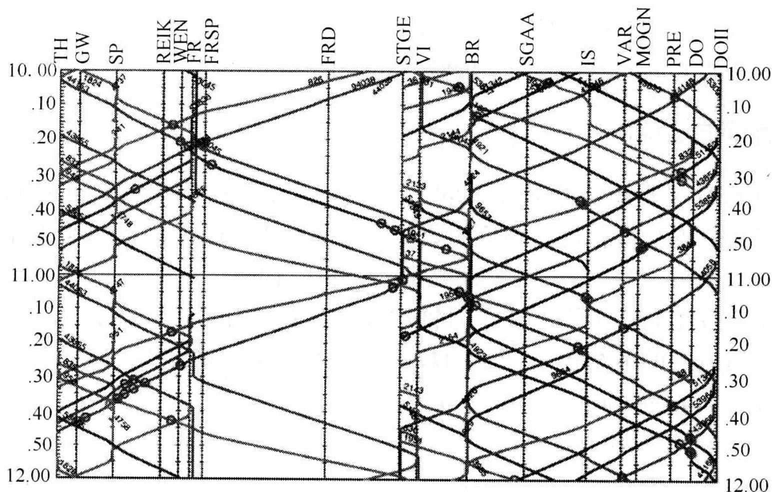



图 2.11 TDD 展示的铁路交通（H. Brändli，IVT，ETH Zürich）© ETH Zürich

对比图2.9与图2.11，两者的关键区别在于图2.9并不能反映真实时间。UML的设计初衷也并不包含对实时应用的支持，但UML2.0已经包含了时序图（Timing Diagrams）功能，它允许与真实时间进行关联。这样的UML"剖析"功能，允许对时间进行更多的注解，[Martin and Müller，2005]、[Müller，2007]。

TDD很适合于对一些典型调度进行描述，但SC与TDD却不能对模型中的同步信息进行很好地描述。如在2.10所示的例子中，无从得知列车在布鲁塞尔的相遇，是出于巧合，或者是为了同步调度的需要。同时，也很难从图中表达的时序（最小/最大时序行为）中得到允许误差。

## 2.4　通信有限状态机

如果希望在更清晰的层次来描述SUD，则需要更加精确的模型。在本章开始，就提到了需要描述状态机行为。状态图是状态机行为描述的经典方式，图2.12（与图2.1相同）展示了经典状态图的例子，即有限状态机（Finite State Machine，FSM）。

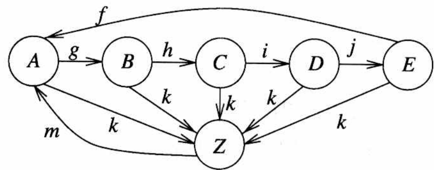



图2.12　状态图

在图2.12中，圆圈代表着不同的状态。如果假定FSM中只有一个状态是处于活动态，这样的FSM称为**确定性**（Deterministic）FSM。边表示状态的改变，边上的符号表示事件。假定FSM中有多个状态处于活动态，从边引出的方向上，发生了边上的符号事件，则FSM将从当前的活动态转变到边所指向的终点状态。有些FSM是受时钟驱动的，这样的FSM称为同步FSM（Synchronous FSM）。对于同步FSM，只有在时钟跳变时，状态才能发生改变。FSM同样可以产生输出（未在图2.12中体现）。对于经典的FSM，更多内容请参考如Kohavi［Kohavi，1987］等人的著作。

### 2.4.1　时间自动机

传统的FSM不提供与时间相关的信息，为了在模型中包含时间参数，传统的FSM进行了相应的功能扩展。时间自动机（Timed Automata）在本质上是自动扩展为带实际值的变量。"变量反映了系统中的逻辑时钟数，它们的值在系统初始化时都是零，而后按相同的节奏增长。时钟约束，如对时钟沿的控制，常用于控制自动机的行为。当时钟的值与边上标明的数值一致时，状态将发生改变，此时时钟可能被复位为零"［Bengtsson and Yi，2004］。

图2.13展示了这样一个例子。

应答机在启动后即处于最左边的状态，当有呼叫信号时，时钟$x$复位为0，应答机转入等待状态，此时被叫者提起受话器即可开始对话，而后挂机。如果被叫者

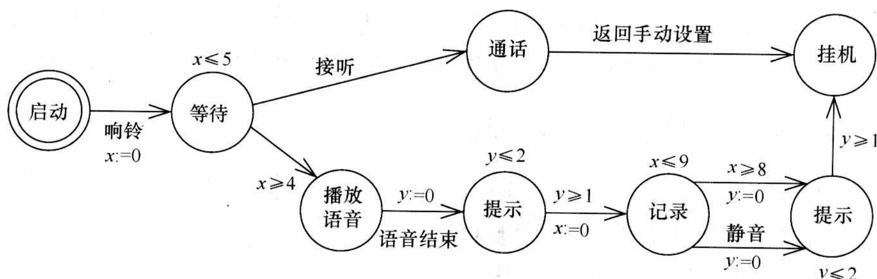



图 2.13　电话应答机中的呼叫服务示例

在经历时间值 4 后仍然未接听，则转入语音播放状态。

在进入语音播放状态后，将播放一段以提示音为结束的语音信息。时间 $y$ 保证了提示音至少会播放一次，而后时间 $x$ 再次复位到 0，同时应答机进入准备记录状态。如果时间值到达 8，或者呼叫者一直没有操作，则再播放一次提示音，同样至少需要播放一次。此后进入终止状态，在本例中，状态的改变可能是由于输入导致的（如接听），或者是因为**时间约束**（Clock Constraints）引起的超时。

时间约束描述了一些有可能发生的状态迁移，但它们也有可能不会发生。为了确保状态的变化，需要定义一些**局部常量**。本例中使用了 $x \leqslant 5$，$x \leqslant 9$ 和 $y \leqslant 2$ 这样的局部常量，从而保证在使能条件为真后，状态的转换会在限制的时间之内产生。一般只会使用一个时间约束条件，仅在作一些证明问题时才使用两个时间约束。

时间自动机的正式定义如下［Bengtsson and Yi，2004］：

假设 $C$ 是表示时间的非负实数集合，$\Sigma$ 是有限的输入集合。

**定义：时间约束**可以用公式表达：$x \circ n$ 或 $(x-y) \circ n$，其中 $x, y \in C$，$\circ \in \{\leqslant, <, =, >, \geqslant\}$ 同时 $n \in \mathbb{N}$。

虽然时间可以是实数，但此处使用的 $n$ 必须是整数。扩展到有理数也很简单，只需要经过简单的乘法即可。假定 $B(C)$ 是时间约束的集合。

**定义**［Bengtsson and Yi，2004］：**时间自动机**是这样的一个元组 $(S, s_0, E, I)$，其中：

1）$S$ 是有限状态的集合；

2）$s_0$ 是初始状态；

3）$E \subseteq S \times B(C) \times \Sigma \times 2^C \times S$ 是边的合集，$B(C)$ 是一系列组合场景，$\Sigma$ 是转换发生需要的输入条件，$2^C$ 是当转换发生时会被重置的时间变量；

4）$I: S \rightarrow B(C)$ 是每个状态的常量集合，$B(C)$ 是对某个特定状态 $S$ 必须保持的常量，这些常量以公式的形式描述。

时间自动机常常需要用第一个定义来辅助说明，它有很多时间因素，因此很难理解。关于时间自动机的更多资料，可以参考 Dill 等人［Dill and Alur，1994］与 Bengtsson 等人［Bengtsson and Yi，2004］的系列论文。

时间自动机在传统的自动机的基础上，增加了更多的时间信息。但是时间自动机却并不能完全满足人们对特定实现的要求，尤其是它不能提供层次性与并发性的架构描述。

## 2.4.2 状态图：隐性共享内存通信

状态图是结合了自动机原理的典型语言，它同时也支持分层与并行设计，但它在对时间的描述方面能提供的方式很有限。

状态图由 David Harel [Harel，1987] 在1987年首先进行了介绍，而后又进行了更加准确的描述 [Drusinsky and Harel，1989]。根据 Harel 的说法，状态图这一名称的选取，来源于它是“惟一一种没有将图表与流程或状态结合的方式”。

### 2.4.2.1 分层架构的建模

状态图语言是对 FSM 的扩展，扩展的关键点即**层次化**（**Hierarchy**）。状态图可以用于对基于状态机的行为进行建模。

我们可以**超状态**（Super-States）来描述分层架构。

定义：

1）包含多个其他状态的 $S$ 为**超状态**。

2）被包含的子状态称为超状态的**子状态**（**Sub-State**）。

图2.14是一个状态图的示例，它是图2.12的分层架构图示。

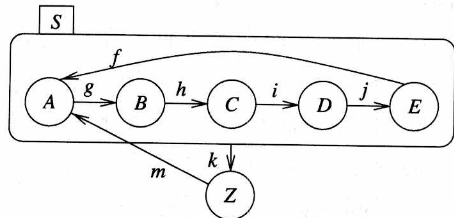



图2.14　层次化的状态图

超状态 $S$ 包含了 $A$、$B$、$C$、$D$、$E$ 5个子状态。假定 FSM 处在状态 $Z$ [$Z$ 也被称为**活动态**（**Active State**）]：如果在 FSM 中加上输入 $m$，则 $A$ 与 $S$ 将成为新的活动态。如果 FSM 处在状态 $S$，此时加上输入 $k$，无论 FSM 是处在 $S$ 中的 $A$、$B$、$C$、$D$ 还是 $E$，$Z$ 都将成为新的活动态。在本例中，组成超状态 $S$ 的所有子状态都是非分层架构的，但其实 $S$ 中的子状态仍然可以再包含自己的子状态。**当超状态中的任一子状态处于活动态时，超状态也处于活动态。**

定义：

1）一个不再包含其他状态的状态，称其为**基本态**（**Basic State**）；

2）对于基本态 $S$，其超状态被称为**父始态**（**Ancestor State**）。

在图2.14中，FSM 在任意时间只能处于超状态 $S$ 的子状态之一，这样的父状态称为**OR 超状态**[⊖]。

⊖ 更精确地，它们应该被称为 XOR 超状态，因为 FSM 可能存在于 $A$、$B$、$C$、$D$、$E$ 之一上，但此术语在本书中极少使用。

在图 2. 14 中，$k$ 可以被认为是一种异常，从而状态从 $S$ 处发生了跳转。图 2. 14 也展示了状态图可以对异常信息进行较为紧凑的描述。

状态图可以对包含多个子系统的分层系统进行描述，这些子系统仍然可以包含更小的子系统。整个系统的层次被称为树（Tree），树根相当于整个系统，内部的节点相应于分层的描述符（在状态图中被称为超节点），系统树上的叶子则相应于非分层描述符（称为基本态）。

现目前为止，一直使用明确而直接的边来表示下一个状态，这样做的缺点是无法对外屏蔽超状态的内部结构。但在真正分层架构中，又常常需要这样做，使超状态内部的改变不会影响到其他架构。当然，使用其他一些机制来描述下一个状态是可以解决这个问题的。

第一种补充机制是**默认状态机制**（Default State Mechanism），它可以被用于超状态，从而表示当超状态处于活动态时，特定的子状态也将处于活动态。在状态图中，默认状态以一个小的实心圆来表示，实心圆其自身不包含任何状态。图 2. 15 展示了默认状态机制的使用，它与图 2. 14 是等同的。

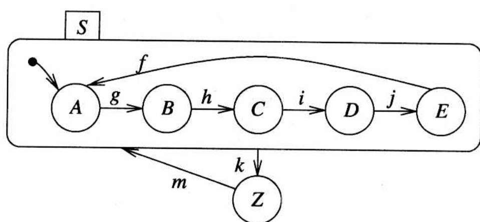



图 2. 15　使用默认状态机制的状态图

**历史机制**（History Mechanism）是指定下一个状态的另一种机制。它可以在超状态跳转前，较容易地返回到上一个处于活动态的子状态。历史机制使用一个带有圆圈的字母 H 来表示。为了表示一个超状态的最原始状态，历史机制常与默认机制结合使用。图 2. 16 展示了这样的一个例子。

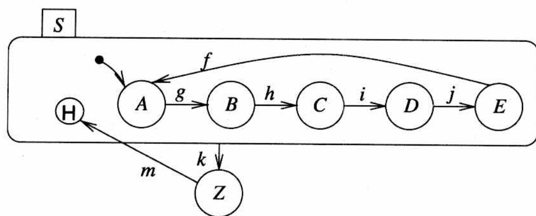



图 2. 16　使用默认状态与历史状态机制的状态图

现在，FSM 的行为就有一些不一样了。当系统正处于 $Z$ 时加上输入 $m$，如果是第一次进入 $S$，则 FSM 会进入状态 $A$，否则它将进入上次从 $S$ 跳转时的状态。这种机制有很多应用，如使用 $k$ 来代表异常，就可以使用输入 $m$ 来返回异常发生时的系统状态。状态 $A$、$B$、$C$、$D$、$E$ 可以将 $Z$ 视作一个子程序，当完成这个“子程序”$Z$ 后，将返回到调用子程序的状态。

图 2. 17 对图 2. 16 进行了重新描述，在此处结合使用了默认状态机制与历史机制。

实践中，还需要能方便地对系统的并发性进行描述的技术。为了解决这一问题，状态图语言提供了第二种超状态，称为AND 超状态。

**定义**：如果系统包含的状态 $S$ 在其所有子状态中一直存在，则称超状态 $S$ 为**AND 超状态**。

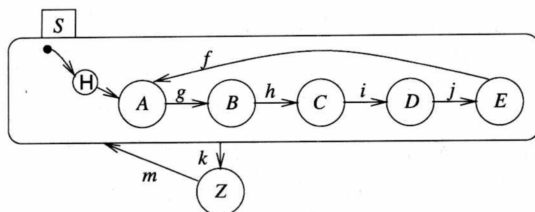



图 2.17　综合历史与默认状态机制的图

以包含 AND 超状态模型的电话应答机为例，如图 2.18 所示。

电话应答机一般需要并行完成两个任务：监听呼入线路、等待用户按键操作。在图 2.18 中，这两个相应任务的状态为 Lwait 与 Kwait。被呼叫后则进入状态 Lproc 进行处理，同时在 Kproc 状态中等待用户的按键操作。在系统最初状态时，假定开/关按键的状态（一般对应着 on 与 off 键）是不一样的，此时按键操作不会进入 Kproc 态。在 "off" 被按下时，只有在重新按 "on" 后，才会重新进入呼入线路监听状态与按键等待状态。此时，系统也进入默认的 Lwait 与 Kwait 状态。当开关处于 "on" 时，系统将一直处于呼入线路监听状态与按键等待状态。

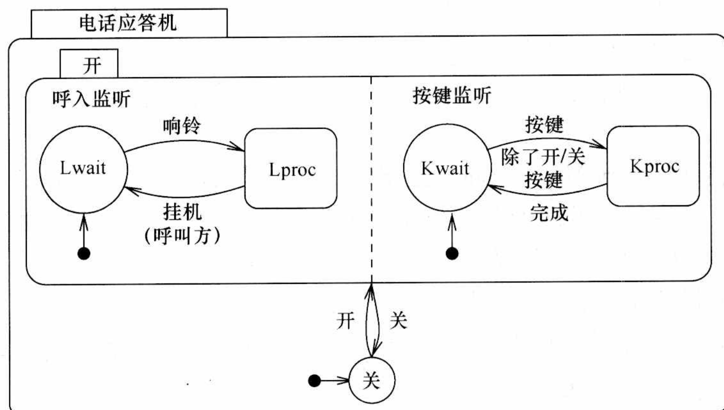



图 2.18　电话应答机

对于 AND 超状态，因其子状态的改变而触发的事件都是可以明确定义的。这些子状态可以是历史状态与默认状态的组合，并且子状态会发生明确的转换。即使子状态之中只有一个发生了转换，所有的子状态都需要全部重新进入，理解这一点非常重要。相应地，从 AND 超状态中跳转到另一状态时，所有的子状态也将发生跳转。

假如修改一下电话应答机，使其开/关按键像其他开关一样，也由 Kproc 状态进行解码（见图 2.19）。

开/关按键的操作由 Kwait 进行检测，假定首先进入状态 Kproc，而后进入 off 状态，这种转换也将导致系统从呼入监听状态跳转。重新打开应答机将导致系统重

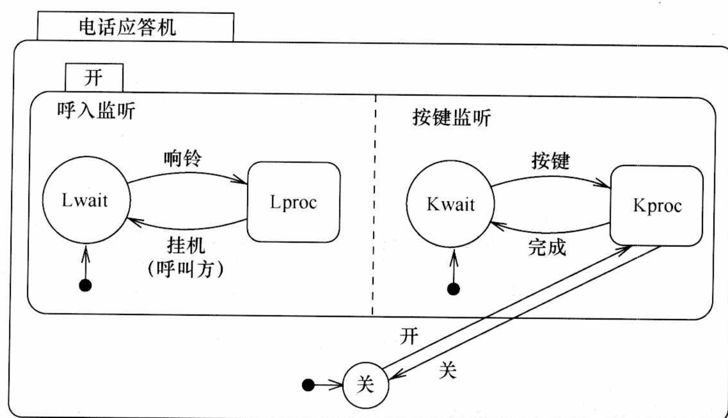



图 2.19　电话应答机经修改的开/关按键处理流程

新进入呼入监听状态。

AND 超状态以状态图的方式提供了对并行处理进行描述的关键机制。每个子状态都可以看作独立的状态机，它们之间彼此通信，组成了**通信有限状态机**（CF-SM）。这个术语已经被用作了本章的标题。

小结一下，状态图中的状态机可以是 AND 状态、OR 状态或基本状态之一。

#### 2.4.2.2　定时器

出于对嵌入式系统中时间进行建模的需要，状态图也提供了对定时器进行描述的方法。定时器使用图 2.20 中的符号（左侧）进行表示。




图 2.20　状态图中的定时器

系统在某个状态经过定时器指定的时间后，即发生超时，而系统将退出当前状态。定时器同样也可以被用于分层架构的设计。

在电话应答机中，定时器可用于在较低层次上去描述 Lproc 的行为。图 2.21 展示了 Lproc 的可能行为。这里指定的时序与图 2.13 有微小的区别。

在图 2.18 中，对于如呼叫方挂机的异常情况，一旦呼叫方挂机，则 Lproc 也同时终止。对于来自被叫者的挂机行为，Lproc 的状态设计可能会引起一些不便：如果被叫者先挂机，则电话将失灵（同时静音），一直到呼叫者也挂机。

状态机语言包含了许多其他语言元素：完整的资料可以参考 Harel 的研究［Harel，1987］。关于状态机语言的详细语法，可以参考 Drusinsky 与 Harel［Drusinsky and Harel，1989］的相关著作。

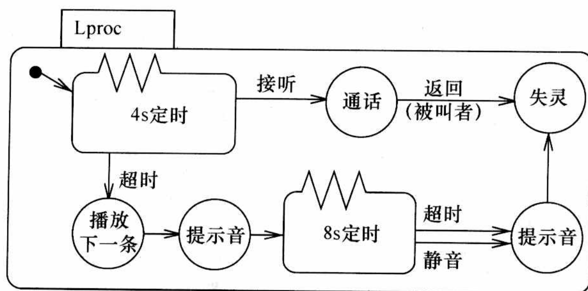



图 2.21　Lproc 对呼入的处理

#### 2.4.2.3　标识符与 StateMate 语义

目前为止，还没有考虑过基于扩展的 FSM 产生的输出，这些输出可以使用标识符来进行指定。标识符的格式一般是“event [condition] /reaction”，这三部分都是可选的。reaction 描述了 FSM 对状态转换的反应，如产生一些事件，或者对一些变量赋值。condition 部分描述了变量处于何值以及系统当时处于何种状态。event 部分描述了发生的事件。事件可以在系统内部或外部产生，内部事件是状态转换的结果，它用标识符中的 reaction 部分进行反应。外部事件则常常在对系统外部环境的模型中进行描述。

例如：

1）on-key/on：=1（测试事件与变量赋值）；

2）[on =1]（测试条件需要的变量值）；

3）off-key [not in Lproc]/on：=0（测试事件，某个状态的测试条件，变量赋值。当事件发生，并且测试条件也满足时，将发生变量的赋值）。

StateMate 是一种商业化的状态图实现，标识符在 StateMate 的语义全文 [Drusinsky and Harel，1989] 中有相应讲解。StateMate 假定系统是按描述符所表示的步骤进行的，每一步都包含 3 个阶段：

1）在此阶段，主要是评估外部的一些改变对场景及事件的影响，它也包含对依赖于外部事件的某些功能的评估。这个阶段不包含任何状态改变。在所给出的简单示例中，这个阶段并不是必须的。

2）此阶段主要是预测所需要作出的转换，评估对变量的赋值，但新的值仅赋给临时变量。

3）在第 3 阶段，状态改变开始生效，并且变量将使用新的值。

阶段 2 与阶段 3 的分开对保证 StateMate 模型的可回溯性至关重要。考虑如图 2.22 所示的 StateMate 模型。

在第 2 阶段，$a$ 与 $b$ 的新值都是首先存储在临时变量中，记为 $a'$ 与 $b'$。在最后阶段，临时变量的值才复制到了用户定义的变量中：

阶段 2：$a'$：$=b$；$b'$：$=a$；

阶段 3：$a := a'$；$b := b'$。

当事件 $e$ 发生时，两个变量的值将发生交换。这种行为类似于连接到相同时钟源（见图 2.23）的两个交叉耦合寄存器（每个变量各一个），它反映了这两个寄存器[⊖]的同步（从时钟上考虑）有限状态机操作。

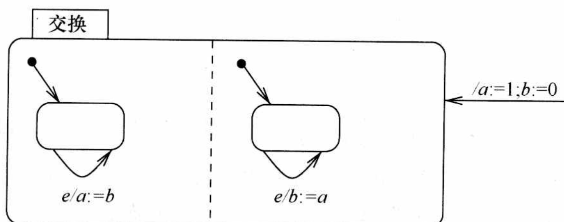



图 2.22 依赖性的赋值

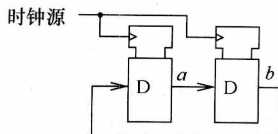



图 2.23 交叉耦合的 D 型寄存器

如果这些操作的各个阶段不是分开的，则两个变量可能会被赋予相同的值，这个值取决于赋值操作的时序。对于试图反映出硬件同步操作的语言，将操作分成两个（至少）阶段是很典型的情况。在 VHDL 中，这种划分更常见。基于这种划分，仿真结果将不会依赖于模型中的哪部分先被执行到，这种特性非常重要。否则，相同的仿真可能会产生多个似乎是正确的结果，这在设计流程中常常会导致混乱。对于具有确定行为的实际电路仿真，这种混乱绝对不是人们期望得到的。

对于这种特性，有多种命名：

1）Kahn［Kahn，1974］称此特性为有序性。

2）在其他一些论文中，此特性被称为确定性。这个术语也有多个不同的含义：

① 此术语可被用于表示非确定性的有限状态机，它们在同一个时刻可能会存在多个状态［Hopcroft et al.，2006］。

② 可能有非确定性操作符的语言，对于这些操作符，不同的行为也会被执行。

③ 对于系统行为依赖于某些不明确输入的系统，一些作者常称它们具有非确定性。

④ 在此处 Kahn 使用术语“确定性（Determinate）”。

在本书中，选择了使用 Kahn[⊜]的方式，从而减少了可能的混淆。只有在那些不确定的行为不会发生时，StateMate 模型才具有确定性。如在某些情况下，转换中的冲突是可以接受的（见图 2.24）。

考虑图 2.24a，如果系统处于左侧的状态，发生了事件 $A$，需要明确此时将发

---

⊖ 对于本书中的门与寄存器，采用的是 IEEE 标准的电路符号［IEEE，1991］。图 2.23 展示的是受时钟驱动的 D 型寄存器。

⊜ 在本书第 1 版中，使用了术语“确定性”，同时补充了一些解释。

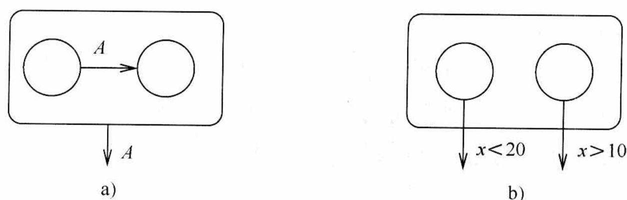



图 2.24 StateMate 转换中的冲突

生什么样的转换。如果使用随机方式来解决这种冲突，则系统的行为就是非确定性的。通常，使用优先级机制可以避免这一类型的冲突。现在再看图 2.24b，在 $x=15$ 时仍然会产生冲突，这样的冲突有时候很难被觉察到。为了实现确定的行为，这就需要完全消除系统中以随机方式解决的冲突。

有很多场景需要明确的描述非确定性的行为（如从两个输入中选择其中之一）。在这类场景中，确实描述那些在运行时发生的选择（参见 2.8.2 节关于 ADA 的状态选择）。

相对于分层架构的状态，StateMate 更能准确地描述确定的行为。这种实现与基于软件的分层实现相对应，在这种实现中，所选择的方案并不被明确描述。

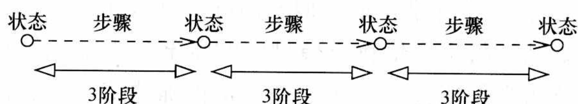



图 2.25 StateMate 模型在执行中的步骤

2.4.2.3 节一开始描述的 3 个阶段需要被反复的执行，每一次执行称为一个步骤（step，见图 2.25）。

假定每当有事件发生或者变量改变时，均会执行一个新的步骤。这些变量以及所产生的（当前时间产生的）事件，在 StateMate 模型中被称为状态[⊖]。在执行完第 3 个阶段后，将得到一个新的状态。对步骤的解释，使人们能更准确地定义事件的语义。这些所产生的事件，都可以来自内部或是外部。**对于当前正在执行的步骤来说，因它而产生的后续事件对它是不完全可见的**。假如，在某个时钟跳变时在使能寄存器中永久性地置位某个 bit，它又对下一次时钟跳变的计算结果产生影响，它们都不是永久性的事件。

与之相对应，变量保持着它们的值，直到被重新赋值。根据 StateMate 的语法，新的变量值在每一个步骤变化中，对模型的其他部分都是完全可见的。也就是说，StateMate 语法会把两个相邻的步骤间的变量赋值操作告知给其他部分，它明确地使用广播机制来同步变量的变化。这意味着针对基于共享内存的平台设计，以及基

⊖ 更多地使用了术语“状态机”来代替“状态”，事实上，术语“状态机”在 StateMate 中有不同的含义。

于消息传递的分布式系统而言，状态图或 StateMate 更适用于前者。这种语言从一开始，即使没有明确的指示，它也假定系统是基于共享内存通信。对于分布式系统，在两个步骤之间更新所有的变量是非常困难的事情。由于使用这种广播机制，StateMate 并不适合对分布式系统进行建模。

#### 2.4.2.4　评估与扩展

以控制功能为主的局部系统，是状态图的主要应用领域。它的突出优点是可以随意进行层次的嵌套，以及使用 AND 和 OR 状态机；其次，StateMate 语法已经有非常规范的定义［Drusinsky and Harel，1989］；第三，当前已经有很多基于状态图的商业化工具，如 StateMate［IBM，2010a］与 StateFlow［MathWorks，2010］。这其中的许多工具还可以将状态图转化为等效的 C 或 VHDL 语言。VHDL 语言在经过综合后，即可转化为硬件。因此，可以认为基于状态图的工具，已经提供了完整的从状态图规范，直至硬件产生的设计流程。它也可以产生可编译与执行的 C 程序，提供了基于软件意识的设计方法与可能性。

不幸的是，这种自动转换的效率仍然令人担忧。如可以将 AND 状态机的子状态映射为 UNIX 的进程，这在低端处理器上必然不会是有效的应用实现。试图在状态图中得到高效的编程实现，这是不太可能的，因为状态图不是基于对象的。另外，广播机制也使它并不适合分布式系统。状态图并不包含可描述复杂计算的结构，它也很难描述硬件结构与非功能性的行为。

状态图的商业实现通常都提供一些机制，从而消除模型中的一些限制。如 C 语言可被用于表达一些程序结构，而 StateMate 中的模块图（module charts）可以用于表示硬件结构。

状态图提供了超时机制，但没有指定更简明的方式去描述其他关于时序上的需求。

UML 包含了状态图的变种，它允许对状态机进行建模。在 UML 的版本 1.0 中，这些图被称为状态图，在 2.0 及以后的版本中被称为状态机图。在 UML 中关于状态机图的语义与 StateMate 不同，即它不再包含仿真的 3 个阶段。

### 2.4.3　同步语言

#### 2.4.3.1　动机

就状态机图而言，描述复杂的 SUD 非常困难，它不能用于表达复杂的计算。标准的编程语言可以表达复杂的计算，但多线程的执行顺序很难预测。在可抢占式的多线程运行环境中，不同的计算之间可能存在多种交叉关系。理解并发系统中的所有可能行为是一件非常困难的事情，其关键因素之一就是多线程有可能存在多种执行顺序，而不同的执行顺序可能产生不同的结果。非确定性的系统行为将产生多种负面效应，如对一些设计中的问题进行验证。如果分布式系统的时钟不可靠，则很难得到确定的系统行为。但对于非分布式系统，可以避免因不必要的非确定语义

引起的问题。

对于同步语言，有限状态机与编程语言被归并为同一模型。同步语言可以用于表达复杂的计算，但最根本的执行模型还是有限自动机，它们描述了并行的自动操作。基于以下的关键特性可以得到确定的行为：“…当自动机的构成是并行的，自动机的“同时”转换将引起系统的转换”［Halbwachs，1998］。这也意味着，如果自动机受其自己的时钟控制，则可以不再考虑自动机状态变化的不同顺序。可以假定存在单一的全局时钟，每个时钟节拍将重新检测所有输入，从而计算出新的输出与状态并作出转换，这就需要模型间各个模块有快速的广播机制。理想的并发情况是能保证系统的确定行为，这也是相对于一般通信有限状态机（CFSM）模型的限制，因为在 CFSM 中的每个 FSM 都有其自己的时钟。同步语言也体现了同步硬件的操作理论，以及如 IEC 60848［IEC，2002］和 STEP 7［Siemens，2010］一类的控制语言。参考 Potop-Butucaru 等人［Potop-Butucaru et al.，2006］的著作可以更进一步地了解同步语言。

#### 2.4.3.2 典型的同步语言：ESterel、Lustre 和 SCADE

Esterel［Esterel Technologies Inc.，2010］、［Boussinot and de Simone，1991］和 Lustre［Halbwachs et al.，1991］已经将保持系统的确定行为作为其语言的重要特性。

Esterel 是一种激励语言：当系统因输入事件的发生而处于活动时，Esterel 模型将产生输出事件。Esterel 是一种同步语言：所有的激励输出都假定在零时间内完成，Esterel 有能力分析系统在离散时间内的行为。理想化的模型应该是可以完全消除关于时间段重叠，以及某次事件在上一次事件未能完成之前到来而引起的冲突。与其他一些并行语言一样，Esterel 也有并行的操作符，记作“‖”。与状态图相似，它也使用基于广播机制的通信，但它的通信都是即时的，这又与状态图不一样。在此处，它的意思是“在相同的时钟周期内”。这意味着在特定的某个时间段内产生的信号，在模型的各个部分都能观察到它，如果某个模块对此信号敏感，也必须在相同的时间段内作出反应。为了达到稳定的状态，也许需要多次反复。关于 Esterel 的更多更新信息，可以参考 Esterel 的主页［Esterel Technologies Inc.，2010］。

Esterel 与 Lustre 使用了不同的语法去描述 CFSM。Esterel 使用了命令式的语言方式，而 Lustre 看起来更像是数据流语言（参考 2.5 节关于数据流的描述）。SyncCharts 就是 Esterel 的图形化版本之一。这三种语言都是基于 CFSM 进行阐述的，它们有着许多相似相关性。商业化的 SCADE［Esterel Technologies，2010］包含了这三种语言的特点，它主要用作一些对安全性有苛刻要求的软件模块设计，如空中客车（Airbus）。

基于 StateMate 中的 3 个仿真阶段，StateMate 有着同步语言的关键特性，并且如果消除了其中的冲突，则它也是确定性的。根据 Halbwachs 的描述，“除了不具

备即时广播的特性外，StateMate 是很完备的一种同步语言”［Halbwachs，2008］。

## 2.4.4　SDL：消息传递的场景

### 2.4.4.1　语言特性

状态图并不适合对分布式的通信有限状态机建模。对于分布式系统，消息传递是更好的通信机制。因此，将展示两个例子：其中之一是基于通信有限状态机；另一个是基于异步消息传递。

SDL（Specification and Description Language，规范描述语言）就是为分布式应用而设计的。它产生于 20 世纪 70 年代，在 80 年代发布了正式的语法。SDL 是一种国际电信联盟（International Telecommunication Union，ITU）指定的标准语言，它的第一版标准 Z. 100 在 1980 年发布，而后在 1984 年、1988 年、1992（SDL-92）年、1996 年与 1999 年进行了更新，其相关的标准包括 SDL-88、SDL-92 与 SDL-2000［SDL Forum Society，2010］。

一些用户会倾向于选择图形化的规范语言，但另一些人却更愿意选择文本化的语言。SDL 则提供了文本化与图形化两种兼容方式。进程是 SDL 的基本组成要素，它代表了扩展的有限状态机的模型，这种扩展也包含了对数据的操作。图 2.26 展示了 SDL 的图形化表达方式中的图形符号。




图 2.26　SDL 图形化方式中使用的符号

在本例中，将思考如何使用 SDL 对图 2.27 中的状态图进行表达。除了增加输出，删除了状态 Z 及修改了受影响的信号 $k$ 以外，图 2.27 与图 2.15 是一模一样的。

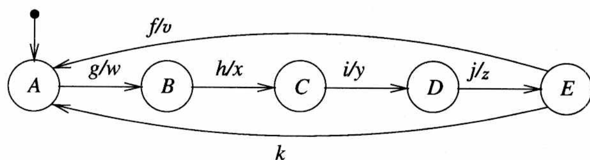



图 2.27　SDL 描述的 FSM

图 2.28 是图 2.27 相应的图形化 SDL 表达。

很明显，图 2.28 中的表达与图 2.27 中的状态图表达是等价的。

作为 FSM 的扩展，SDL 的进程可以对数据进行操作。在 SDL 的进程中，可以声明局部的变量，它们可以预先定义，又或者在 SDL 描述时定义。SDL 还支持抽象的数据类型（ADT），在 SDL 中声明与操作的语法与在其他语言中很相似。图 2.29 展示了如何声明、赋值与判断。

SDL 还包含了诸如程序一类的编程语言要素，程序调用也可以以图形化的方式

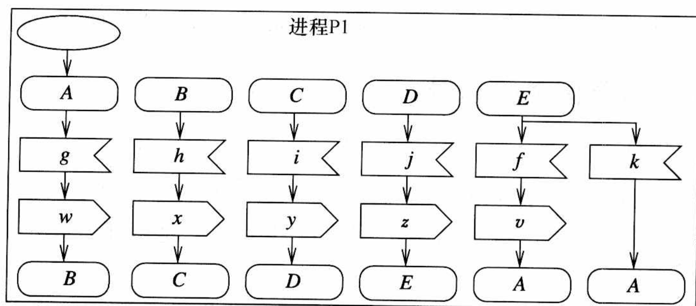



图 2.28　图 2.27 的 SDL 表达

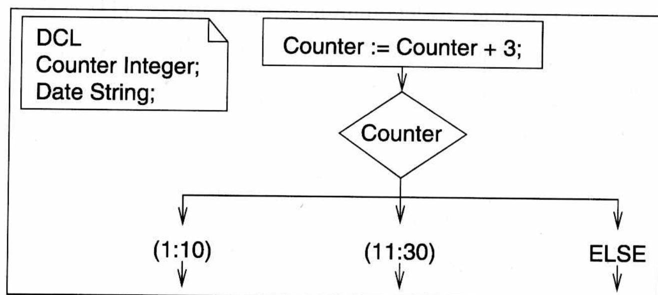



图 2.29　SDL 中的声明、赋值与判断

进行表达。在 SDL-1992 中，发布了基于对象的编程特性，而后又在 SDL-2000 中得到了扩展。

扩展的 FSM 仅仅是 SDL 描述符中的基本要素。通常，SDL 描述符还要包含多个接口进程，或者 FSM。进程可以向其他进程发送消息。SDL 中进程间通信的语义是基于异步消息传递的，每个进程都有一个与之相关联的先进先出（FIFO，First-In First-Out）队列。发送到一个特定进程的消息都将存储到相应的 FIFO 队列中（见图 2.30）。

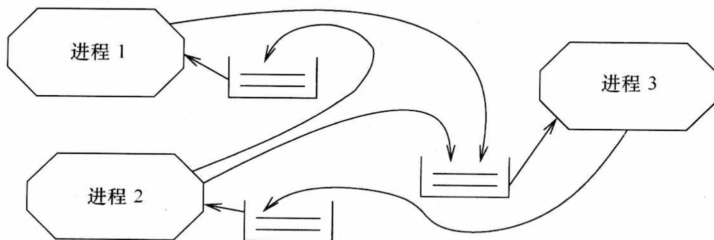



图 2.30　SDL 的进程间通信

假定每个进程都会从 FIFO 队列中获取可用的下一条消息，同时检查它是否与当前所描述的输入状态相匹配。如果匹配，则产生相应的状态转换以及输出；如果

来自 FIFO 队列中的消息与所列举的输入不匹配，此消息将被忽略（除非使用了存储机制）。理论上，假定 FIFO 队列的长度是无限的，这就意味着对于 SDL 模型语义的描述符，可以不必考虑 FIFO 的溢出问题。然而在现实系统中，无限长度的队列是不可能存在的，它必须有一定的长度。这就是 SDL 的问题之一：为了从规范中得到实现，必须为 FIFO 队列提前规定一个合适的长度上限。

使用进程间通信图，可以更好地观察哪些进程之间存在通信。进程间通信图包含用于发送与接收消息的通道。在 SDL 的场景中，术语“消息（signal）”表示了模型化自动机的输入与输出。

例如，图 2.31 展示了进程间通信图 B1，它有 Sw1 和 Sw2 两个通道，括号内包含了相应通道传输的消息名称。

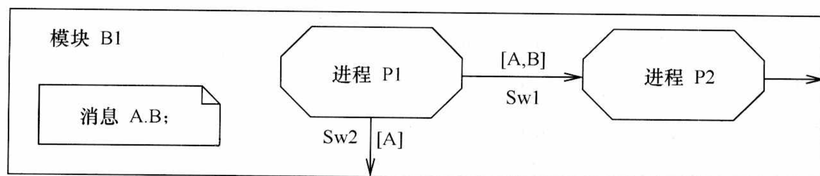



图 2.31　进程间通信图

有 3 种方式来表示消息的接收：

1）**通过进程标识**：在图形化的输出符号中使用接收进程的标识（见图 2.32 左）。






图 2.32　描述消息的接收

并不需要在编译阶段就固定进程的数量，因为进程完全可以在运行时动态创建。OFFSPRING 就是一个进程动态创建的子进程的消息标识。

2）**显性的**：直接标识出通道的名称（见图 2.32 右），Sw1 就是通道的名称。

3）**隐性的**：当消息的名称暗含着通道名称时，消息将使用这些通道。如在图 2.31 中，消息 B 暗示着它总是通过通道 Sw1 进行通信。

在进程中不可以再定义新的进程（进程不可以嵌套），但它们可以按层次组织起来，称为**模块（Blocks）**，最高层的模块即**系统**。进程间通信图是模块图中的一个典型例子。进程间通信是分层描述符中基本单元层级的行为，B1 可被用于进程间的通信模块（如在图 2.33 中的 B）。

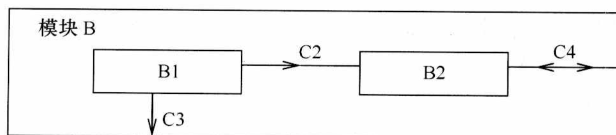



图 2.33　SDL 模块

在分层结构的最高层便是系统（见图2.34），如果外部环境同样被模型化为一个模块，则系统在其边界上不会有任何通道与之相连。

图2.35展示了基于图2.31、图2.33以及图2.34的分层模型。

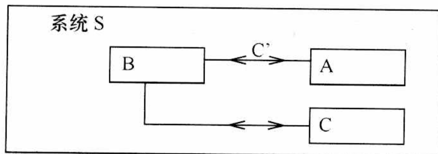



图2.34　SDL系统

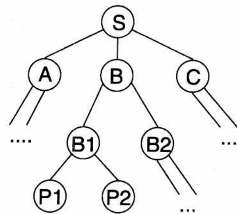



图2.35　SDL分层结构

进程间通信图与分层描述符的树叶很贴近，系统描述符则表示了它们的树根。在SDL-2000的版本中，消除了在分层建模时的很多限制。使用SDL-2000，模块与进程的描述能力得到了进一步协调，它使用了更通用的代理概念（agent concept）。

为了支持对时间的建模，SDL还包含定时器（timers）。定时器可以在进程内部进行定义，可以分别使用SET与RESET原语来设置与复位它们。

图2.36展示了定时器T的使用。

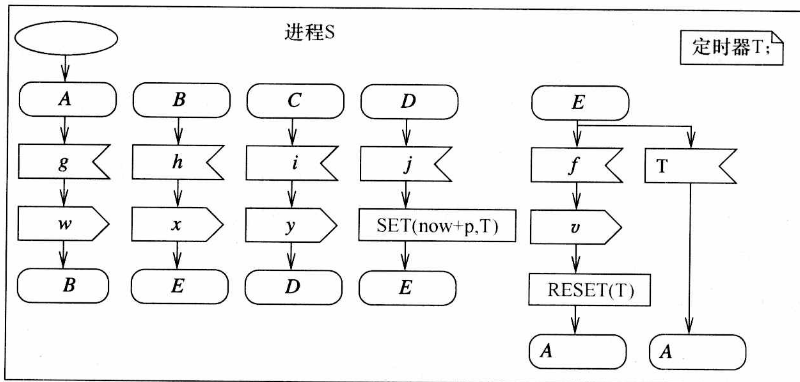



图2.36　使用定时器T

图2.36中的定时器与图2.28相似，在状态从$D$转换到$E$这一过程中，定时器被设定为当前时间加上$p$。对于从$E$至$A$的转换，使用了$p$个时间单位的超时定时器。如果在消息$f$被接收到之前定时器超时，则状态转至$A$，但不产生输入信号$v$。

SDL可用于描述计算机网络中的协议栈。图2.37展示了通过一个路由器连接的3个处理器，处理器与路由器之间基于FIFO进行通信。

处理器与路由器基于分层协议（见图2.38）。

每一层都在更加抽象的层次上来描述通信机制。每一层的行为通常以有限状态机来进行建模。这些FSM的描述细节都依赖于网络协议，可能会非常复杂。通常，

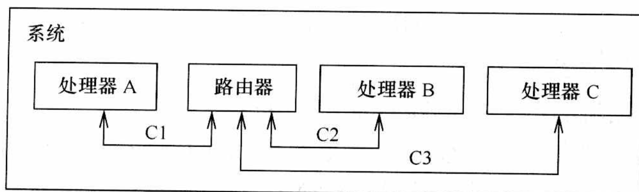



图 2.37 在 SDL 中描述的小型计算机网络

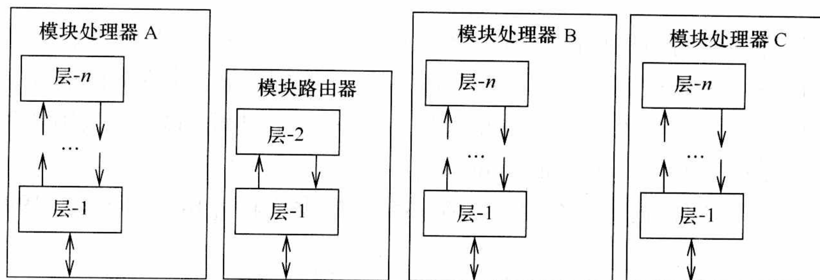



图 2.38 SDL 中的协议栈

这些行为包含校验与处理错误，整理与分发信息包。

SDL 工具包括 UML 接口、SC。在 SDL 论坛上有完整的工具列表 [SDL Forum Society，2009]。

Estelle [Budkowski and Dembinski，1987] 是另一种设计用于描述通信协议的语言。与 SDL 相似，Estelle 也假定通信是通过通道与 FIFO 进行的，但 Estelle 与 SDL 并没有统一为一种语言。

#### 2.4.4.2 对 SDL 的评估

SDL 非常适合分布式应用，它也得到了广泛的应用，如在 ISDN 的设计方面。

SDL 并不需要具有确定性（它的消息并不需要在指定的时间发送到 FIFO 中）。

可靠的实现需要知道 FIFO 的深度上限，但有时候可能很困难。定时器的概念对于软实时要求就已经足够了，但对于硬实时要求还远远不够。

SDL 并不像状态图那样提供对分层设计的支持。

在 SDL 中没有充分的编程支持（最近的版本中已经开始支持），对非功能性的属性也没有对应的描述符。这样的话，即使是它在作为参考模型时非常有用，但看起来 SDL 并没有什么吸引人的地方。

## 2.5 数据流

### 2.5.1 范围

数据流是对现实应用进行描述的一种非常“自然的”方式。数据流模型反映

了数据如何从一个模块到另一个模块的过程［Edwards，2001］。每一个模块都对数据以这样或那样的方式加以处理。

下面是对数据流的定义［Wikipedia，2010］：

**定义**：数据流建模“是标识、模拟与规范数据如何在信息系统中转换的过程。数据流建模检查过程（将数据从一种形式转换为另一种形式），数据存储（保存数据的区域），外部实例（由谁来发送数据，由谁来接收数据），数据流（数据流经的路径）”。

**数据流程序**使用的是直接的图形化方式，其中的节点也被称为执行器，表示了流程中的计算，圆角矩形表示了计算的数据流向。数据流中计算都被认为是功能性的，即它们都依赖于输入的值。数据流图形中的每个进程都被分解为时序化的流程，它们都是原子操作（Atomic Actions）。

如在图2.39中描述了一个语音控制系统的数据流程［Ko and Koo，1996］。

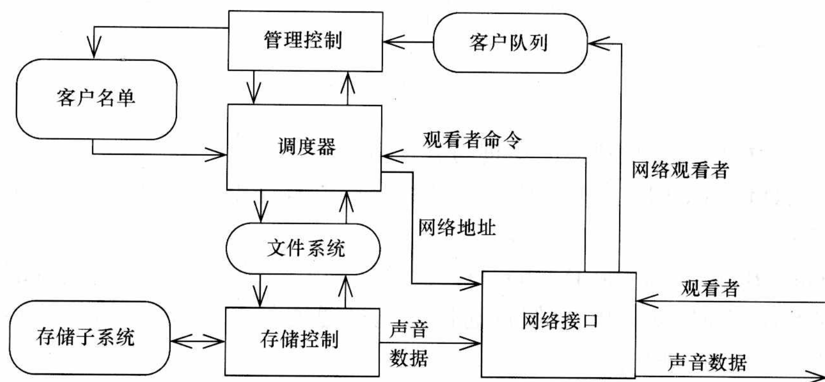



图2.39 语音控制系统

对于无约束的数据流，很难描述系统的属性。因此，一般都只对有约束的模型进行描述。

## 2.5.2 Kahn处理网络

Kahn处理网络（Kahn Process Networks，KPN）［Kahn，1974］是一种特定的数据流模型。与其他数据流模型一样，KPN也包含着节点与边。其中，节点对应着由程序或任务执行的计算。KPN流程图与所有数据流程图相似，展示了所进行的计算以及它们之间的依赖关系，但KPN流程图并不能展示出各个计算之间必须遵守的顺序（这一点与冯·诺依曼编程语言中的规范相反，如C语言）。KPN中的边展示了通过含有有限FIFO的通道进行的通信。计算时间与通信时间也许没有可比较性，但总是需要保证通信在一定的时间内完成。由于已经假定FIFO的大小总是能满足需求，因此通信时的写操作都是非阻塞性的。读操作总是从单一通道进行

读取，在尝试读取之前，操作节点并不知道数据是否能成功读取，而进程不能因为此操作而阻塞。但如果是从一个空的 FIFO 队列中读取，则会被阻塞。只允许单一进程从特定的队列中读取，同时也只允许单一进程向一个队列中写。因此，如果输出数据被发送到了多个进程，这些进程必须对数据进行复制。进程间的通信仅有通过 FIFO 队列这一种方式。

在下面的例子中，p1 与 p2 分别从对方接收数据，p2 对数据进行加操作，而 p1 对数据进行减操作：

```
process p1(in int u, out int v){
    int i;
    i = 0;
    for (;;) {
     send(i,v);   -- send i via channel v
     i = wait(u); -- read i from channel u
     i = i-1;
    }}
  process p2(in int v, out int u){
   int i;
   for (;;) {
    i = wait(v);
    i = i+1;
    send(i,u);
   }}
```

图 2.40 是 KPN 对上例的图形化表达。

很显然，由于消息不会在通道中累积，因此在此例中并不需要 FIFO。Levi 仿真软件 [Sirocic and Marwedel，2007b] 可以对本例及其他一些例子进行仿真。

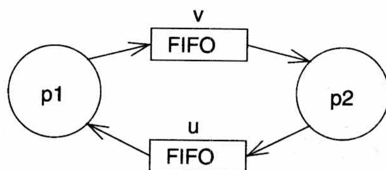



图 2.40　KPN 的图形化表达

约束形成了 KPN 的关键特性：节点从一个通道读取数据的顺序是由读操作的时序决定的，它并不依赖于节点向通道发送数据的顺序。也就是说，读操作的顺序与节点产生数据的速度无关。对于一组给定的输入数据，KPN 总是产生相同的结果。这种特性非常重要，从而在 **KPN 的仿真中，其结果与仿真的速度无关，它们总是相同的**。尤其是对于某些节点与分布式的执行，其仿真结果并不依赖于使用的硬件加速器。这种属性被称为“确定性”，接下来也将使用此种方式。SDL 在 FIFO 上遇到的冲突问题在 KPN 中并不存在。由于有这些属性，KPN 常被用于表达设计流程中

的内部环节。

有时，KPN 也包含了扩展的“归并”的操作（相应于 ADA 的选择（Select）状态）。此操作允许从一系列队列中同时有序地读取数据，同时等待通道产生数据。这些操作都是不确定性的行为：如果两个输入同时到达，其处理顺序是不确定的。这种扩展在实践中有时很有用，但它破坏了 KPN 最重要的属性。

总之，由于估计进程的精确行为比较困难，Kahn 进程在运行时都需要调度机制。这些问题来源于在通道的速度与节点方面并没有作出假设。对于真实的 KPN 模型，在一般的场景中很难确定使用有限长度的 FIFO 是否能满足要求。因此，在早期设计阶段执行时间并不确定时，这种模型就足够进行分析了。有一些可用的调度算法［Kienhuis et al.，2000］可以参考。对于 KPN，进程的数量是固定的，它并不在运行中改变。

## 2.5.3　同步数据流

如果对节点与通道加上时序约束，则调度将明显变得更加容易，关于缓冲区大小的考虑也会更容易判断。同步数据流（Synchronous Data Flow，SDF）［Lee and Messerschmitt，1987］就是这样的一种模型。

SDF 有多种图形化的符号，图 2.41 左图展示了同步数据流的例子。例子中的图形是直接表示图，节点 $A$、$B$ 分别展示了 * 与 + 操作。当有输入时，节点就可以开始计算。如果任意两个节点之间有依赖关系，则需要使用边来连接这两个节点。

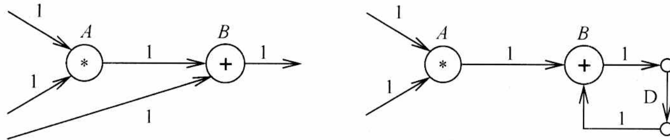



图 2.41　同步数据流的图形化表达

对于每个执行，节点中的计算被称为 firing。对于每个 firing，有许多符号与数据表达被处理及产生。在同步数据流中，一个 firing 中产生与处理的符号都是常量。常量上的标签表示了相应的符号数。这些常量使不同速率的信号处理应用变得更加容易，这些应用中的一些信号计算速率是另一些信号计算速率的整数倍。如在电视机中，某些信号的计算是 100Hz，而某些信号按 50Hz 进行计算。通常，发送到边上的符号数与被处理的符号数是一样的。假定 $n_s$ 是单个 firing 中一些发送者所产生的符号，$f_s$ 是相应的处理频率；$n_r$ 是单个 firing 中的一些接收者所处理的符号，$f_r$ 是相应的处理频率，于是就有

$$n_s * f_s = n_r * f_r \tag{2.1}$$

这种情况在图 2.42 中以另外一种方式进行了展示。如果 $n_s \neq n_r$，则需要缓冲操作，相比于 Kahn 处理网络，FIFO 的大小可以被比较容易地计算出来。

术语**同步数据流**表示符号的输入与符号的处理是按同步方式进行的（在所有时间点）。而术语**异步消息传递**则意味着符号可以被缓存到 FIFO 中。基于处理与产生固定数量符号这一物性，可以较容易地在编译阶段就确定执行时序以及对内存的需求。因此，应该尽量避免在运行时有复杂的调度。SDF 图可以表示出处理中的延时，这可以用带连线的符号 D 来表示（见图 2.41 右图）。SDF 图也可以被转换成单任务或多任务系统中周期性的调度（参见［Pino and Lee，1995］）。图 2.41 中展示了一个简单的合理调度的例子，它包含了两个事件序列 $A$、$B$（永久循环的）。事件序列 $A$（$A$，$A$，$B$）（$A$ 的执行次数是 $B$ 的两倍）并不合理，因为这将导致 $A$ 与 $B$ 之间的 FIFO 缓冲区总是会累积出无限的符号。

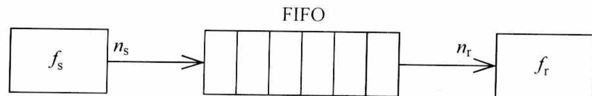



图 2.42　不同处理速率的 SDF 模型

SDF 在有些场景非常有意义，例如在多媒体系统中。在这种情况下，每个符号都相应地包含了一些音频或视频信息，如一个音频帧或视频帧。在处理端，如果使用前面提到的非冯·诺依曼语言对其进行建模，则会遇到很多问题，而如果使用 SDF（见图 2.43）则可以轻松解决，这里也不会存在死锁问题。但是，SDF 并不允许在运行过程中添加新的处理端。

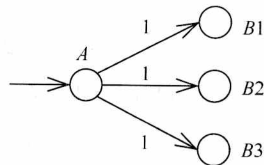



图 2.43　SDF 观察者模式

SDF 是一种确定性的模型，它并不适合对控制流程进行建模，如有多个分支的系统等。不过，最近也有了许多经扩展与变异的 SDF，倾向于这些领域的应用（如参考 Stuijk［Stuijk，2007］）：

1）举例来说，可以依据一个相关的有限状态机的**模式**进行建模。对于每一种模式，使用不同的 SDF 图会非常有用，许多事件将导致这些模式之间的转换。

2）同类同步数据流（Homogeneous SDF，HSDF）图是 SDF 图的一种典型场景。对于 HSDF 图，在每个 firing 产生与处理的符号总是 1。

3）对于静态循环数据流（Cyclo-Static Data Flow，CSDF），每个 firing 产生与处理的符号可以是变化的，但必须是周期性的。

包含控制流的复杂 SUD 在建模时必须使用更通用的计算图结构。

### 2.5.4　Simulink

计算图结构通常用于控制工程中。在这个领域中，MATLAB 的 Simulink（仿真）工具箱［The MathWorks Inc.，2010］、［Tewari，2001］使用得非常广泛。MATLAB 是基于数学模型的建模与仿真工具，如偏微分方程等。图 2.44 展示了 Simulink 模型的一个例子［Marian and Ma，2007］。

图 2.44 右侧的运算放大器与饱和器展示了模拟信号的处理模型。在许多场景

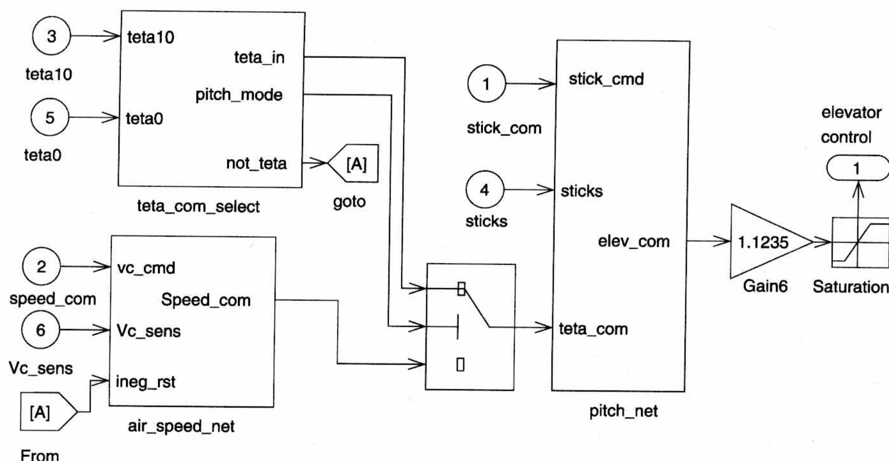



图 2.44　Simulink 模型

中，“电路”均包含一些模拟元件符号，如积分器、差分器等。图中间的开关说明 Simulink 同样也可以对某些控制流进行建模。

这种图形是一种比较直观的表达方式，它允许控制工程更专注于控制功能，而不需要考虑为实现这些功能所需的代码。对于设计中涉及的模拟电路，仍然建议使用传统的电路符号来表达。图形化的重要目标之一，即为了从这些模型中综合出软件，它常与术语**模型化设计**相关，但这个术语并没有精确的定义。

Simulink 模型的语法反映的是对数字计算机系统的仿真，它也能反映出类似的模拟电路的行为，但有可能与模拟电路的真实情况不太一致。那么，Simulink 模型的语法应该如何定义呢？Marian 与 Ma［Marian and Ma，2007］对其语法有如下定义：“对于模块（节点）的执行与通信过程，Simulink 使用了一种理想化的时序模型。在仿真时间中，节点的执行与通信都是非常迅速的过程；其次，仿真中的时间按精确的步长增长”。也就是说，模型总是按时间一步一步执行。在每一步的执行中，对节点进行运算处理（在该步的零时间点），而后将新的值送到与之相连接的其他输入端。这种解释中并没有指出时间步长的具体值，由于即使是缓慢变化的输出也有可能被频繁地重新计算，它也没有立即指出如何将系统进行软件实现。

Simulink 非常适合对诸如汽车、火车之类的实际系统进行高层建模，同时对这些系统的行为进行仿真。MATLAB 与 Simulink 也能很方便地对数字信号处理系统进行建模。为了产生具体的实现方案，MATLAB/Simulink 模型都必须首先被转换为一种硬件或软件设计系统支持的语言，如 C 或 VHDL。

Simulink 模型中的组件提供了**执行器**（Actors）的特殊场景。可以认为执行器一直在等待输入，一旦它需要的输入事件产生，则立即执行与它们相应的操作。SDF 是基于执行器语言的另一种场景，在这种语言中，并不需要像在非冯·诺依曼语言中那样将控制传递给其他执行器。

# 2.6 Petri 网

## 2.6.1 简介

Petri 网是一种计算过程图模型，它可以对控制流程进行非常充分的描述。事件上，Petri 网模型仅仅控制着流程之间的依赖关系，它也需要模型中的数据来进行扩展，它更强调因果依赖。

在 1962 年，Carl Adam Petri 发布了他关于对因果依赖进行建模的方法，也就是后续的 Petri 网 [Petri，1962]。Petri 网不假定系统之间存在着全局同步，因此它特别适合于描述分布式系统。

**条件**（Conditions）、**事件**（Events）与**流关系**（Flow Relation）是 Petri 网的主要元素：条件可能被满足或不被满足，事件可能发生，流关系描述了事件发生必须要满足的前提条件，它也描述了事件发生后会变成真的条件。在 Perri 网中的图形表示中，通常使用圆形来表示条件，方框来表示事件，边来表流关系。图 2.45 展示了 Petri 网的一个例子。

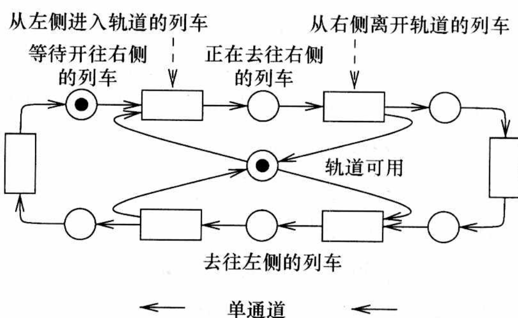



图 2.45　单轨道铁路段

图 2.45 展示了单轨道铁路段控制不同来车方向的互斥操作，它使用了令牌来避免相反方向来车的冲突问题。在上面的 Petri 网中，令牌表示为模型中间的一个条件符号，它以部分填充的圆（圆的中间还包含着另一个实心的圆）来表示必需条件（即轨道可用）。同时，列车准备从左向右行驶这一条件（在图 2.45 中也使用一个部分填充的圆来表示），对于“来自左侧的列车进入轨道”的事件，这两个条件都必须满足，称这两个条件为**前置条件**（Preconditions）。只有事件的前置条件已经满足，事件才可能发生。在事件发生后，令牌将立即失效，即没有列车等待着进入轨道。因此前置条件就不再被满足，也就没有了表示为必须条件的部分填充的圆（见图 2.46）。

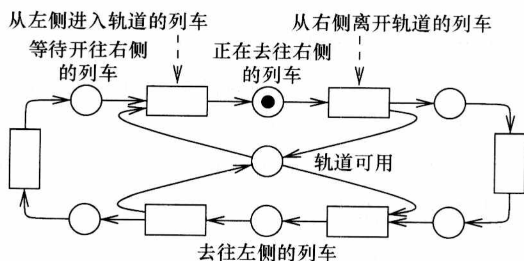



图 2.46　使用“轨道”资源

但是，现在有列车从左向右行驶，因此相应的条件又被满足了（见图 2.46）。在事件发生之后被满

足的条件，被称为后置条件（Postcondition）。通常，如果一个事件的所有前置条件都为真（或满足）了，则事件就可以发生。如果事件发生，则前置条件将不再被满足，而后置条件生效。边上的箭头标识了一个事件的前置条件与后置条件。仍然基于这个例子可以知道，当列车离开轨道时，它将把令牌归还给模型中间的条件（见图2.47）。

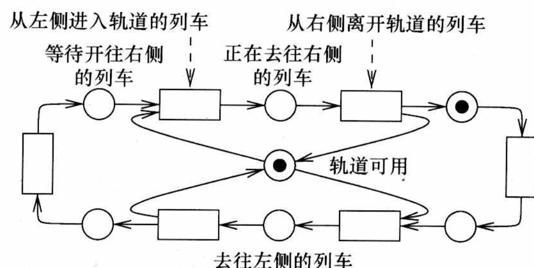



图2.47　释放“轨道”资源

如果两辆列车竞争单向轨道路段的使用权（见图2.48），则它们中只有其一能进入轨道。

在这种场景中，下一个被激发的状态是非确定性的。在进行网络分析时，必须充分考虑可能发生的情况。对于Petri网，更乐意于使用它对非确定性系统进行建模。

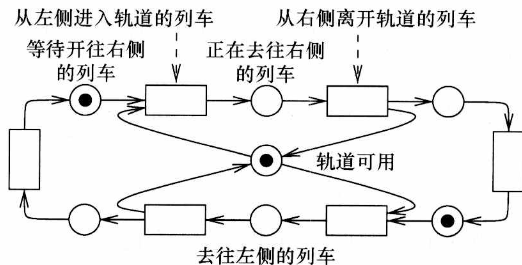



图2.48　“轨道”资源的冲突

Petri网的重要优点之一，就是它可以用于证明系统的属性，它是产生这些证明的一种标准化方式。为了证明这一点，需要对Petri网进行更正式的定义。考虑三类Petri网：条件/事件网（Condition/Event Nets）、库所/变迁网（Place/Transitions Nets）以及预测网（Predicate Transition Nets）。

## 2.6.2　条件/事件网

首先对Petri网中的条件/事件网进行更正式的定义。

定义：$N=(C, E, F)$ 被称为一个网，满足以下条件：

1）集合 $C$ 和 $E$ 不相交；

2）$F\subseteq(E\times C)\cup(C\times E)$ 是二元关系，也称作流关系。

集合 $C$ 被称为条件，而集合 $E$ 被称为事件。

定义：设定 $N$ 是一个网，同时 $x\in(C\cup E)$，那么有

1）$^{\bullet}x:=\{y\mid yFx, y\in(C\cup E)\}$ 为 $x$ 的**前集**（Pre-Set），如果 $x$ 表示一个事

件，则$^{\bullet}x$也称为$x$的**前置条件**（Preconditions）集合；

2）$x^{\bullet}:=\{y \mid xFy, y\in(C\cup E)\}$ 称为$x$的**后集**（Post-Set），如果$x$表示一个事件，则$^{\bullet}x$也称为$x$的**后置条件**（Postconditions）集合。

如果这些集合表达的事件$\in C$，即，如果$x\in E$，则更倾向于使用术语前置条件与后置条件。

**定义**：设定（$c$，$e$）$\in C\times E$

1）如果$cFe\wedge eFc$，则（$c$，$e$）被称作一个**循环**（Loop）；

2）如果$F$不包含任何循环，则$N$被称为**纯网**（Pure Net）（见图 2.49 左图）。

**定义**：如果转换$t_1$与$t_2$的前置与后置条件不一样，则$A$被称为**简单网**（见图 2.49 中间图与右图）。

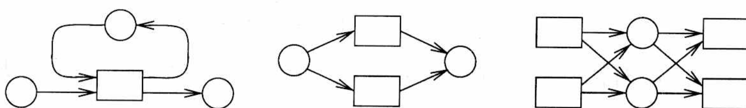


图 2.49 非纯网（左图）与非简单网（中间图与右图）

含有不满足某些附加约束的非隔离元素的网，被称为**条件/事件网**（图中的节点是两个不相关的集合）。鉴于接下来更多的讨论是关于 Petri 网的通用类，因此此处不过多讨论这些附加约束。

## 2.6.3 库所/变迁网

对于条件/事件网，每一个条件只有一个令牌。但对于许多应用，它们的每个条件都可能有多个令牌。网络中允许一个条件有多个令牌的情况，被称为库所/变迁网。事实上，库所是与当前讨论的条件相对应的，而变迁与事件相对应。每个库所的令牌数被称为一个**标识**（marking）。从数学上讲，按定义允许有些库所的容量为$\omega$，标识可以为每个库所指定有限多个资源。

假定$\mathbb{N}_0$表达的是含 0 的自然数，可以将库所/变迁网更正式地定义如下：

**定义**：（$P$，$T$，$F$，$K$，$W$，$M_0$）被称为一个库所/变迁网$\Leftrightarrow$

1）$N=(P, T, F)$是库所$p\in P$的网，变迁$t\in T$，流关系为$F$；

2）映射：$K$：$P\rightarrow(\mathbb{N}_0\cup\{\omega\})\setminus\{0\}$ 表示了库所的容量（$\omega$表示无限的容量）；

3）映射：$W$：$F\rightarrow(\mathbb{N}_0\setminus\{0\})$ 表示了边的权重；

4）映射：$M_0$：$P\rightarrow\mathbb{N}_0\cup\{\omega\}$ 表示了边的初始标识。

边的权重会影响变迁发生所需要的标识数量，同时它也指出了当变迁发生时产生的标识数量。假定$M(p)$表示了库所$p\in P$的当前标识，$M'(p)$表示的是变迁$t\in T$发生后的标识。属于前置条件的边的权重表示了从前集的库所中移除的令牌数。相应地，属于后置条件的边的权重表示了在后集的库所中添加的令牌数。标识

$M'(p)$ 通常按如下进行计算：

$$M'(p)=\begin{cases}M(p)-W(p,t), & \text{如果 } p\in {}^{\bullet}t\backslash t^{\bullet}\\ M(p)+W(t,p), & \text{如果 } p\in t^{\bullet}\backslash {}^{\bullet}t\\ M(p)-W(p,t)+W(t,p), & \text{如果 } p\in {}^{\bullet}t\cap t^{\bullet}\\ M(p) & \text{其他条件}\end{cases}$$

图2.50展示了变迁 $t_j$ 如何影响当前标识的例子。

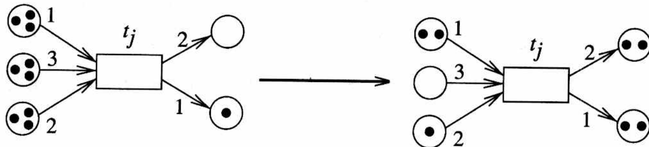



图2.50　新标识的产生

默认情况下，没有标签的圆环其权重为1，没有标签的库所的容量 $\omega$ 是无限的。

现在来解释变迁 $t\in T$ 必须满足的两个条件：

1）对前集中的所有库所 $p$，令牌的数据至少要与从圆环 $p$ 到 $t$ 的权重相等；

2）对后集中的所有库所 $p$，其容量必须要能够容纳 $t$ 产生的新令牌。

满足以上两个条件的变迁被称为M-activated，它的定义如下：

定义：变迁 $t\in T$ 被认为是M-activated $\Leftrightarrow$

$$(\forall p\in {}^{\bullet}t: M(p)\geqslant W(p,t))\wedge(\forall p'\in t^{\bullet}: M(p')+W(t,p')\leqslant K(p'))$$

这些激活的变迁都可以发生，但并不是必须的。当多个变迁处于激活态时，它们发生的时序是非确定性的。

自激变迁 $t$ 影响的令牌数量，可以以一个与 $t$ 相关的向量表达式来定义如下：

$$\underline{t}(p)=\begin{cases}-W(p,t), & \text{如果 } p\in {}^{\bullet}t\backslash t^{\bullet}\\ +W(t,p), & \text{如果 } p\in t^{\bullet}\backslash {}^{\bullet}t\\ -W(p,t)+W(t,p), & \text{如果 } p\in {}^{\bullet}t\cap t^{\bullet}\\ 0 & \text{其他条件}\end{cases}$$

产生于自激变迁 $t$ 的新令牌数 $M'$，对所有库所 $p$ 都可以按如下计算：

$$M'(p)=M(p)+\underline{t}(p)$$

使用“+”来表示向量的加法，则可以将上面的公式重写为

$$M'=M+\underline{t}$$

所有向量的集合$\underline{t}$来自于关联矩阵$\underline{N}$，向量$\underline{t}$是$\underline{N}$的列向量：

$$\underline{N}: P\times T\rightarrow \mathbb{Z};\quad \forall t\in T: \underline{N}(p,t)=\underline{t}(p)$$

使用矩阵$\underline{N}$可以更正式地描述系统的属性。如对于自激变迁不会改变令牌数量总和的库所［Reisig，1985］，可以计算它的集合，这种集合被称为**恒定库所**（Place Invariants）。为了寻找恒定库所，假定 $t_j$ 为单次变迁，当 $t_j$ 发生时，寻找库

所中的集合 $R \subseteq P$，它们的令牌数将不会发生变化。对于这些集合，它们必须满足

$$\sum_{p \in R} \underline{t}_j(p) = 0 \tag{2.2}$$

图 2.51 展示了当变迁发生时，它的令牌总数不会发生变化的情况。

现在介绍库所集合 $R$ 中的向量$\underline{c}_R$的属性：

$$\underline{c}_R(p) = \begin{cases} 1 & \text{如果 } p \in R \\ 0 & \text{如果 } p \notin R \end{cases}$$

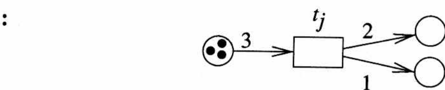



图 2.51　恒定令牌数的变迁

基于这个定义，重写式（2.2）如下：

$$\sum_{p \in R} \underline{t}_j(p) = \sum_{p \in P} \underline{t}_j(p) * \underline{c}_R(p) = \underline{t}_j \cdot \underline{c}_R = 0 \tag{2.3}$$

式中　·——数积。

现在来查找任何变迁的自激均不会改变令牌总数的库所的集合。这意味着对于所有的变迁 $t_j$ 均需满足下式：

$$\begin{aligned} \underline{t}_1 \cdot \underline{c}_R &= 0 \\ \underline{t}_2 \cdot \underline{c}_R &= 0 \\ &\cdots \\ \underline{t}_n \cdot \underline{c}_R &= 0 \end{aligned} \tag{2.4}$$

使用关联矩阵的逆，式（2.4）可以整合成下式：

$$\underline{N}^{\mathrm{T}} \underline{c}_R = 0 \tag{2.5}$$

式（2.5）表达的是一个线性齐次方程。矩阵$\underline{N}$表示的是 Petri 网中节点的权重。式（2.5）中的系统的解向量必然是一个特征向量，其中的元素必然为 1 或 0（如果使用令牌总和的权重，则也可以使用整数权重值）。相对于实数解向量的线性方程，这种情况更加复杂。然而，从式（2.5）中也可以得到其他信息。使用这种证明方法可以展开论证，例如对共享资源如何做到了正确的互斥访问等问题。

来看一个规模更大些的例子，仍然考虑列车同步问题。尝试对运行在阿姆斯特丹、科隆、布鲁塞尔与巴黎之间的高速 Thalys 列车进行建模。在从阿姆斯特丹与科隆到布鲁塞尔的部分路段，列车不会有任何冲突，因此这些路段直接相连而后到巴黎。在从巴黎返回的途中，它们在布鲁塞尔断开了连接。假定 Thalys 列车必须在巴黎与其他列车同步，其相应的 Petri 网如图 2.52 所示。

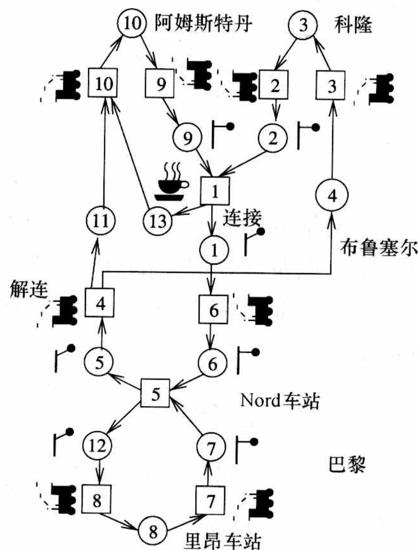



图 2.52　运行在阿姆斯特丹、科隆、布鲁塞尔与巴黎之间的 Thalys 列车模型

库所3与10分别是列车在科隆与阿姆斯特丹等待的模型。变迁2与9是列车从科隆与阿姆斯特丹开往布鲁塞尔的模型。在列车到达布鲁塞尔后，库所2与9将执有令牌。变迁1连接了两辆列车。茶杯状的符号表示某列列车的司机，他将在布鲁塞尔休息，而其他司机仍将开往巴黎。

变迁5是巴黎Nord车站的列车同步情况模型，也有一些将Nord车站与其他车站（例子中使用Nord车站，事实上它也许是巴黎一个非常复杂的车站）相连的列车。当然，Thalys列车不会使用蒸汽发动机，它只是比现代的高速列车更易于比喻。图2.53展示了此例中的矩阵$\underline{N}^{\mathrm{T}}$。

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ | $p_6$ | $p_7$ | $p_8$ | $p_9$ | $p_{10}$ | $p_{11}$ | $p_{12}$ | $p_{13}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $t_1$ | 1 | -1 | | | | | | | -1 | | | | 1 |
| $t_2$ | | 1 | -1 | | | | | | | | | | |
| $t_3$ | | | 1 | -1 | | | | | | | | | |
| $t_4$ | | | | 1 | -1 | | | | | | 1 | | |
| $t_5$ | | | | | 1 | -1 | -1 | | | | | 1 | |
| $t_6$ | -1 | | | | | 1 | | | | | | | |
| $t_7$ | | | | | | | 1 | -1 | | | | | |
| $t_8$ | | | | | | | | 1 | | | | -1 | |
| $t_9$ | | | | | | | | | 1 | -1 | | | |
| $t_{10}$ | | | | | | | | | | 1 | -1 | | -1 |

图2.53　Thalys列车例子中的$\underline{N}^{\mathrm{T}}$

例如，第2行表示激励$t_2$会将令牌$p_2$的数量加1，也表示它会将令牌$p_3$减1。使用线性代数分析方法，可以得到这个系统线性公式的向量解为

$$c_{R,1}=(1,1,1,1,1,1,0,0,0,0,0,0,0)$$

$$c_{R,2}=(1,0,0,0,1,1,0,0,1,1,1,0,0)$$

$$c_{R,3}=(0,0,0,0,0,0,0,0,1,1,0,0,1)$$

$$c_{R,4}=(0,0,0,0,0,0,1,1,0,0,0,1,0)$$

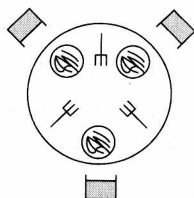
以上向量分别表示了从科隆、阿姆斯特丹驶出的列车，从阿姆斯特丹驶出的列车的司机，以及经过巴黎的列车。可以看到，此轨道上经过的列车与司机数量是固定的（这也正是人们期望的）。这个例子展示的恒定库所，它们可以以标准规范来提供系统属性。

### 2.6.4　预测/变迁网

当实例规模比较庞大时，条件/事件网与库所/变迁网的规模也会变得非常庞大。减小网络大小的常用可行方法，是使用预测/变迁网。将使用“哲学家就餐问题”来证明这一点。这个问题假设有多个哲学家围绕一个圆形的餐桌就餐，在每个哲学家的前面都有一盘意大利面（见图2.54）。

在每个哲学家前面的盘子里只有一把叉子，哲学家要么吃饭，要么思考。每个

哲学家都需要与他相邻两人中某一人的叉子。因此，只有当某位哲学家他的邻座之一不吃饭时，他才能吃饭。


图 2.54 哲学家就餐问题

这个场景可以模型化为一个条件/事件网，如图 2.55 所示。条件 $t_j$ 对应于思考状态，而条件 $e_j$ 对应着就餐状态，条件 $f_j$ 表示有可用的叉子。可以看出，仅仅是这样一个小问题，网络的规模已经变得比较大了。而使用预测/变迁网，就可以减小网络的大小。图 2.56 所示是对相同问题使用预测/变迁网进行的建模。

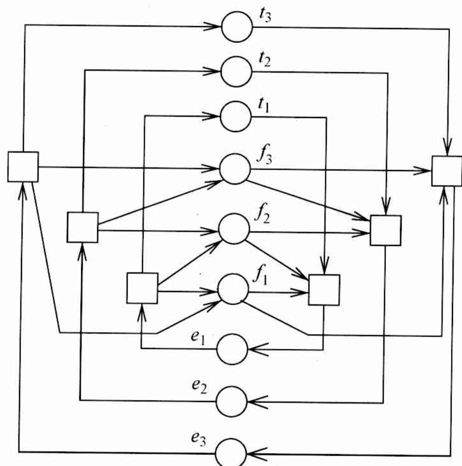


图 2.55 哲学家就餐问题的库所/变迁网模型

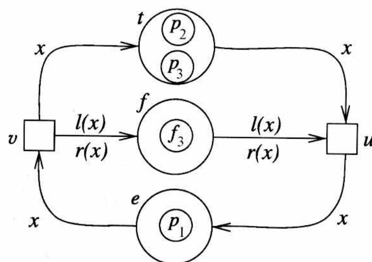


图 2.56 哲学家就餐问题的预测/变迁网模型

在预测/变迁网中，每个令牌都有不同的标识区别于其他的令牌[⊖]。在图 2.56 中，就使用了这种方式来区分哲学家 $p_1 \sim p_3$ 以及叉子 $f_3$。因此，可以使用变量与函数对模块进行标识。在此例中，我们使用变量来区分不同的哲学家，使用函数 $l(x)$ 与 $r(x)$ 分别表示哲学家 $x$ 左侧与右侧的叉子。这两个叉子是变迁 $u$ 的前置条件，也是变迁 $v$ 返回的后置条件。只需要增加更多的令牌，这个例子可以很容易地扩展到哲学家 $n>3$ 的情况。为了与图 2.55 相对应，网络的结构并没有发生变化。

## 2.6.5 评估

Petri 网的关键优点是其对因果依赖进行模型化的能力。标准 Petri 网没有时间概

⊖ 有时也可以考虑令牌使用不同的颜色。

念，在基于对变迁以及它们的前置、后置条件的分析后，立即由本地作出决策。因此，它可以用于地理上的分布式系统进行建模。更进一步，Petri 网有很强的理论基础，它可以简化系统属性的描述。Petri 网并不需要是确定系统，不同的激励时序可以导致不同的结果。Petri 网的描述能力使其构成了其他一些 MoC，包含有限状态机。

在一些场景中，它的长处也同时就是它的缺点。如果模型中需要明确的时间，则就不能使用标准 Petri 网。标准 Petri 网也不能对分层架构进行建模，它也没有编程语言的要素，它很难对数据进行表达。

为了避开这些缺点，现在有了许多扩展版本的 Petri 网。但是并没有一个通用的扩展版本可以满足本章开始提到的所有需求。源于分布式计算的增长，Petri 网也变得非常流行。

包含扩展 Petri 网的 UML 被称为活动图（Activity Diagrams）。扩展包含可以表示决策的符号（像普通的流程图），这些符号的位置与 SDL 类似。图 2.57 展示了这样一个例子。

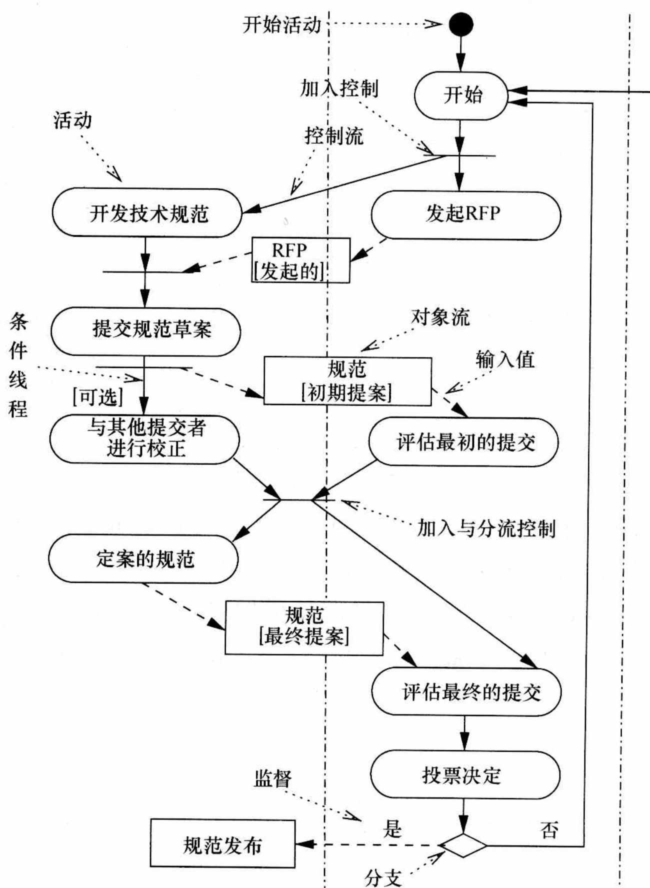



图 2.57　活动图［Kobryn，2001］

这个例子展示了遵守标准的一个流程。控制的分叉与汇聚相应于 Petri 网中的变迁，同时它也使用了最初在 Petri 网中使用的一些符号（水平线）。底部的菱形符号代表了决策。这些活动被分组成“活动通道”（在垂直的点画线区域），因此可以更容易地观察到不同的责任以及文档的交换。有趣的是，像 Petri 网这样的技术在一开始并没有成为主流，直到多年之后，它才因为被 UML 包含在内而得以推广。

## 2.7 基于离散事件的语言

基于离散事件的计算模型是基于事件以及事件处理的仿真都是随时间变化的这一想法。在这个模型中，使用一个队列来存储后续事件，这些事件按它们被处理的先后顺序有序存储。在这种语言的语法中，去掉了对队列中与当前时间有关联的事件，执行相应的行为，同样也可以将新的事件插入到队列中。即使没有可执行的事件，时间也将会增加。

硬件描述语言（Hardware Description Languages，HDL）用于对硬件进行建模，它们通常基于离散事件模型。这里将一直使用 HDL 作为离散事件建模的例子，后续将着重介绍硬件描述语言 VHDL，同时也会简要介绍其他一些 HDL。

软件编程语言与硬件描述语言的最典型区别就是硬件描述语言需要在 HDL 中对时间进行建模，另一个区别是描述不同硬件模块之间的并发性。

### 2.7.1 VHDL

#### 2.7.1.1 简介

VHDL 是 HDL 中的典型代表，它使用进程（Processes）来对并发性进行建模。每个进程是存在并发性的硬件中的一个组件。对于一些简单的硬件组件，也许只需要单独的一个进程就足够了，但对于一些复杂的组件，也许需要多个进程才能对它们的操作进行模型化。进程之间是通过信号（Signals）进行通信的。简单地讲，信号在 VHDL 中对应着物理连接［电线（Wires）］。

VHDL 的起源可以追溯到 20 世纪 80 年代。在那时，大部分的系统设计都使用图形化的 HDL，而使用最广泛的模块是门电路。但是，在使用图形化 HDL 的基础上，仍然可以使用文本化的 HDL。使用文本化语言这一方式的最大好处在于，它们可以比较容易地去表达一些复杂的计算，如包含着变量、循环、函数参数以及迭代的计算。相应地，当数字系统在 80 年代变得更加复杂时，文本化的 HDL 几乎完全取代了图形化的 HDL。文本化的 HDL 在最初只是大学的研究项目，这可以参考 Mermet 等人［Mermet et al.，1998］就当时欧洲地区使用的设计语言的一个调查。MIMOLA 就是其中一种语言，本书的作者也参与了其设计与实现工作［Marwedel and Schenk，1993］，［Marwedel，2008b］。当 VHDL 以及其竞争对手 Verilog 被推广后，文本化的编程语言也开始更加流行。

VHDL 起源于美国国防部（Department of Defense，DoD）的 VHSIC 项目。VHSIC 是 Very High Speed Integrated Circuits[⊖]的缩写。在最初阶段，VHDL（VHSIC 硬件描述语言）的设计是由三家公司完成的：IBM、Intermetrics 与 Texas Instruments 公司。VHDL 的第一个版本发布于 1984 年，而后，VHDL 成为 IEEE 标准，即 IEEE 1076。IEEE 1076 关于 VHDL 的第一个版本发布于 1987 年，而后在 1992 年、1997 年、2002 年与 2006 年进行了更新［Lewis et al.，2007］。由于在语言中支持微分方程方法，VHDL-AMS 允许对模拟系统以及混合信号系统进行建模。由于两种语言均擅长 DoD 的设计，VHDL 在开始时使用了 ADA。由于 ADS 基于 PASCAL，因此 VHDL 有许多语法与 PASCAL 很相似。但是，VHDL 的语法定义更加复杂，并且它的语法不允许有任何歧义。本书，将仅集中在 VHDL 的某些概念上，这些概念在其他语言中也会非常有用。关于 VHDL 的详细讨论不在本书的范围内，VHDL 的标准可以从 IEEE 得到（参考［IEEE，2002］）。

#### 2.7.1.2　实体与结构体

与其他 HDL 一样，VHDL 含有对硬件模块进行并行操作的方法。硬件模块被称为**设计实体**（Design Entities）或**VHDL 实体**（VHDL Entities）。实体又包含了描述并发操作的**进程**（Processes）。根据 VHDL 语法，设计实体由两部分组成：**实体声明**（Entity Declaration）以及一个（或多个）**结构体**（Architectures）（见图 2.58）。

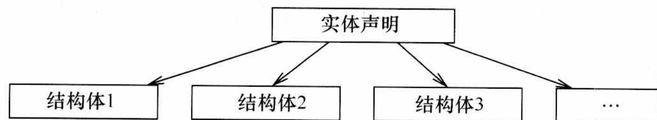



图 2.58　包含声明与结构体的实体

对于每个实体，默认将使用最近被使用最多的结构体，使用其他结构体将另作说明。结构体可以包含多个进程。

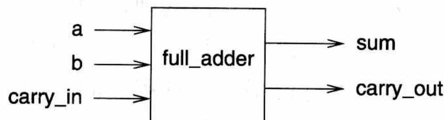



图 2.59　全加器及其接口信号

下面来讨论全加器的例子。全加器有 3 个输入端口及两个输出端口（见图 2.59）。

以下是相应于图 2.59 的一个实体声明：

```
entity full_adder is                   -- 实体声明
   port (a, b, carry_in: in Bit;       -- 输入端口
        sum, carry_out: out Bit);      -- 输出端口
end full_adder;
```

两个连字符（--）后是注释的开始，注释语句一直到整行结束。结构体由结

⊖　网络的设计也是 VHSIC 的一个内容。

构体名及结构体内容构成。需要区分出不同类型的结构体，尤其是组织型（Structural）结构体及行为型（Behavioral）结构体。下面将以全加器为例来分析一下两者之间的差异。行为型结构体包含了足够的信息，从而可以从输入信号与本地状态（如果有的话）计算出输出信号以及输出的时序行为。如下展示了这样的一个例子（< = 表示对信号赋值）：

```
architecture behavior of full_adder is  -- 结构体
begin
  sum       <= (a xor b) xor carry_in after 10 ns;
  carry_out <= (a and b) or (a and carry_in) or
               (b and carry_in) after 10 ns;
end behavior;
```

基于 VHDL 的仿真器可以根据上面描述的全加器输入端的激励，从而显示出输出信号波形。与此相对应，组织型结构体在对实体结构的描述方面显得更加简单。如全加器模型可以使用包含 3 个模块的实体来表示（见图 2.60）。i1 ~ i3 分别是半加器或者门电路。

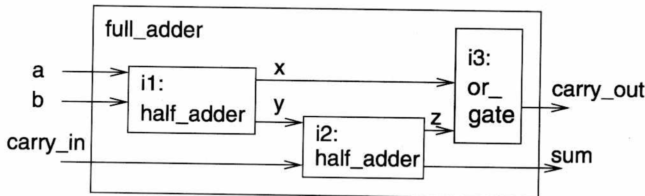



图 2.60　组织型全加器的电路图

在 1987 年版本的 VHDL 中，模块必须首先进行声明。这种声明与其他语言中的声明非常相似（同时它们的功能也是一样的）。即使在 VHDL 的数据库中没有存储关于模块的全部信息（这种情况可能发生在被称为至顶向下的设计中），模块本身也可以提供必需的信息。从 1992 年版本的 VHDL 以后，如果已经在模块库中有相关模块的存储，则并不需要这些声明了。

局部的模块与实体端口之间的连接关系被称为端口映射（Port Maps）。下面的 VHDL 代码表示了图 2.60 中的组织型结构体：

```
architecture structure of full_adder is  -- 结构体开始
  component half_adder
    port (in1, in2: in Bit; carry: out Bit; sum: out Bit);
  end component;
  component or_gate
    port (in1, in2: in Bit; o: out Bit);
  end component;
```

```
 signal x, y, z: Bit;          -- 局部信号
begin                          -- 端口映射段
 i1: half_adder                -- 半加器i1
    port map (a, b, x, y);  -- 端口连接
 i2: half_adder port map (y, carry_in, z, sum);
 i3: or_gate    port map (x, z, carry_out);
end structure;
```

#### 2.7.1.3 VHDL 的进程与赋值

VHDL 将所有上面提到的模块都视为进程。上面使用的关于进程的语法都是简要的介绍，关于进程的语法如下：

```
label: -- 可选
process
  declarations -- 可选
 begin
  statements-- 可选
 end process ;
```

赋值是语句的一种。在 VHDL 中有两种形式的赋值：

1）**变量赋值**：变量赋值的语法为

*variable* := *expression*

无论在什么时候执行到这条赋值语句，表达式都将被重新计算并且变量被立即重新赋值。这种赋值行为与普通编程语言中的赋值语句其实是一样的。

2）**信号赋值**：信号与信号赋值的本意是希望能更准确地对真实硬件系统中的电信号进行描述。与信号相关的值都是瞬时的。在 VHDL 中，这种时间与数值的关系使用波形（Waveforms）来进行表示，从信号赋值中可以计算出波形。关于信号赋值的语法如下：

```
signal <= expression;
signal <= transport expression after delay;
signal <= expression after delay;
signal <= reject time inertial expression after delay;
```

无论此赋值何时被执行，表达式总是被重新计算，并且将计算结果用于波形中的将来时刻。为了计算将来的值，**假定仿真器含有一个事件队列，它存储着在当前时间之后将发生的事件**。队列中的元素按时间进行排序，存储着将来时刻发生的事件（如更新信号）。执行一次信号赋值将在队列中产生一个实体。每个实体都包含着执行事件的时间、受影响的信号以及所赋的值。对于没有包含 after 语句的赋值（第一种语法格式），实体将把当前的仿真时间作为赋值操作发生的时刻。在这种

情况下，赋值的动作将在一个无限小的时刻后发生，称为 δ 延时（参考下面的内容）。这样就使人们在不必改变微小时间的前提下可以更新信号。

对于包含transport 前缀的赋值（第二种语法格式），信号将在延迟指定的时间后被更新。这种格式的赋值被称为**传输延迟模型**（**Transport Delay Model**）。这种模型是基于导线的行为：信号会在导线上（作为一种初步的估计）产生延时。即使是短暂的脉冲信号，它也会在导线上有传输延时。即使传输延时的主要应用是对导线的建模，它也可以用于逻辑电路。假定我们使用传输延时赋值来对一个简单的或-门电路进行建模：

```
c <= transport a or b after 10 ns;
```

即使是短暂的脉冲，第二种语法格式模型也会保证它的发送（见图 2.61）。

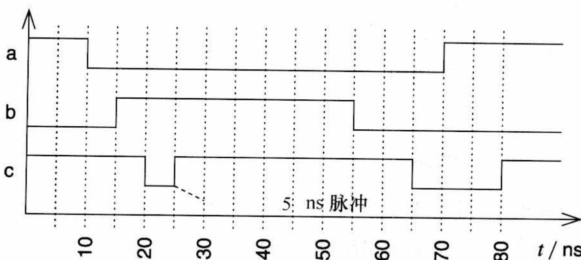



图 2.61　带传输延迟的门电路模型

经历传输延迟的信号赋值时，它将会根据当前计算出的最新时间，删除队列中的所有实体（如果首先以较大的延迟进行了一次赋值，而后又以较小的延迟进行了赋值，则先前较大延迟的对应操作会被删除）。

对于信号赋值中包含 after 但不包含 transport 的语句，都会假定含有**惯性延迟**（**Inertial Delay**）。惯性延迟模型反映了实际电路都会有一些"惯性"，这也就意味着可以抑制信号毛刺的产生。对于信号赋值的第三种形式，小于指定延迟时间的信号改变都将被忽略。对于第四种语法形式，所有小于指定数量的信号改变都将从预期的波形中消除。假设使用一个简单的与门来对惯性延迟进行建模：

```
c <= a or b after 10 ns;
```

对于这样一个模型，中间的信号毛刺就被消除了（见图 2.62）。

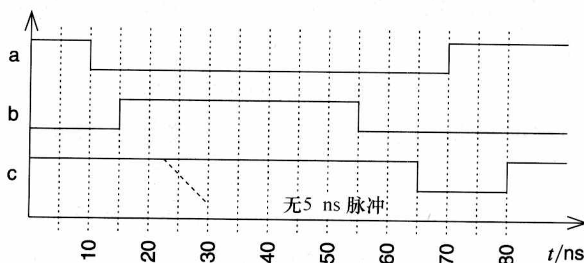



图 2.62　惯性延迟的门电路模型

惯性延迟的实现结果依赖于从预期波形图中移除实体的情况。在这里不过多强调移除的详细原则。

相对于赋值，进程可以有 wait（等待）状态，此状态可将一个进程挂起。下面列举了wait 状态的种类：

1）wait on 信号列表；挂起直到信号列表中有信号发生改变；

2）wait until 条件；挂起直到条件被满足，如 a = "1"；

3）wait for 时间；挂起一段指定的时间；

4）wait；无限挂起。

作为对wait状态的另一种解释，也可以将一些信号添加到进程的开头。在这种例子中，只要列表中的信号发生变化，则进程就被激活。例如，在下面的与门模型中，只要输入信号的值发生变化，则进程的主体将被执行一次，而后再从开始处重新执行。

```
process(x, y) begin
   prod <= x  and y ;
end process;
```

这个模型也等价于

```
process begin
   prod <= x  and y ;
   wait on x,y;
end process;
```

#### 2.7.1.4 VHDL的仿真周期

根据原始标准文档［IEEE，1997］的介绍，VHDL模型的执行可以描述如下："在一个模型的描述中，它的执行由一个初始化阶段以及多个进程状态的重复执行构成。所有进程状态的一次重复被称为一个仿真周期。在此周期中，模型描述中涉及的所有信号的值都会被重新计算。如果重新计算在给定信号上产生了事件，而又有进程状态对此信号敏感，则进程将作为仿真周期的一部分得到执行。"

VHDL的初始化阶段主要是完成信号的初始状态设置，并且每个进程只会执行一次。它在标准里的描述如下[㊀]：

"在初始化阶段开始时，当前时间 $T_c$ 假定为0ns，初始化阶段包含以下步骤[㊁]：

1）对于每一个被明确声明的信号，其驱动值和生效值都被会重新计算，信号的当前值被设置为生效值。生效值是假定在仿真时刻无限之前的一个信号值。…

2）每个…模型中的每个进程在其挂起之前一直执行。…

3）下一次仿真周期的时间 $T_n$（在本例中即第一个仿真周期），它是根据仿真周期的步骤5）计算得出的（见下面的叙述）。"

每个仿真周期从将当前时间设置为必须去考虑状态改变的下一个时刻开始。$T_n$

㊀ 不考虑在VHDL的1997版本中对称为推迟（Postponed）进程中需要明确的信号声明的讨论。

㊁ 为了减小标准中大量细节的影响，部分内容（以"…"代替的部分）在引用时即被省略。

可以在初始化期间计算，或者在仿真周期的最后一次执行过程中进行计算。在当前时间达到它的最大值 TIME′ HIGH 时，仿真即结束。根据原始的标准文档，仿真周期的概念被描述如下："一个仿真周期包含着如下的步骤：

1）当前时刻 $T_c$ 被设置为与 $T_n$ 相等。当 $T_n$ = TIME′ HIGH 并且没有活动的驱动或者在 $T_n$ 时刻没有进程恢复执行，则仿真完成。

2）模型中每一个显式的活动信号都会被更新。（其结果可能是事件。）" …

在当前仿真周期的前一些周期中，将会计算一些信号的后续值。如果 $T_c$ 与这些值变为有效的时刻相对应，则它们被重新赋值。在执行一个仿真周期时，信号的赋值不会立即发生，它们在下一个仿真周期之前均不会发生。信号在改变它们的值时将产生事件，事件可能会使对相应信号敏感的进程得到执行。

3）"对于一个进程 $P$，如果 $P$ 对某一个信号 $S$ 敏感，而在此仿真周期中 $S$ 上又有事件发生，则 $P$ 将重新开始执行。

4）每一个…在当前仿真周期重新开始执行的进程将在它被挂起之前一直运行。

5）下一个仿真周期所需要的时间 $T_n$ 由最早将其设置的如下值决定：

① TIME′ HIGH（仿真的结束时间）；

② 某个驱动变为活动的下一个时间（这是下一个实例的时间，驱动指定的新的值）；

③ 进程开始执行的下一次时间（这个时间由wait for 的状态决定）。

如果 $T_n = T_c$，那下一个仿真周期（如果有）将是一个 $\delta$ 周期。

图 2.63 展示了仿真周期迭代的特点。

$\delta$ 一直是一个有争议的话题。它的目的是在某些即使是用户没有特别指定的情况下，在仿真中引入一个无限小的延时。举一个例子来分析一下这个无限小的延时对触发器的影响。图 2.64 展示了触发器的电路示意图。

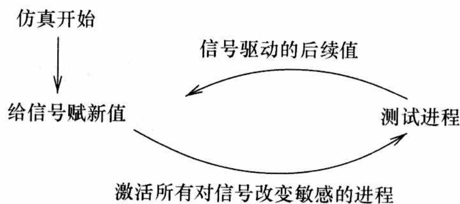



图 2.63　VHDL 的仿真周期

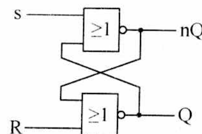



图 2.64　RS 触发器

使用 VHDL 对这一触发器进行建模如下：

```
entity RS_Flipflop is
  port (R: in   BIT;     -- 复位
```

```
        S: in   BIT;      -- 置位
        Q: inout BIT;     -- 输出
        nQ: inout BIT; ); -- 端
end RS_Flipflop;
architecture one of RS_Flipflop is
  begin
   process: (R,S,Q,nQ)
    begin
      Q <= R nor nQ; nQ <= S nor Q;
    end process;
  end one;
```

由于端口 Q 与 nQ 在内部还有读操作，因此它们不能仅仅设置为out，而必须被设置为inout。图 2.65 列出了上面的模型中信号更新的仿真时间。在每一个周期，信号的变换都是通过其中一个门电路传输。在 3δ 之后，仿真即终止。由于 Q 已经是“0”，因此在最后一个周期中没有改变任何信号。

| | < 0ns | 0ns | 0ns+δ | 0ns+2δ | 0ns+3δ |
|---|---|---|---|---|---|
| R | 0 | 1 | 1 | 1 | 1 |
| S | 0 | 0 | 0 | 0 | 0 |
| Q | 1 | 1 | 0 | 0 | 0 |
| nQ | 0 | 0 | 0 | 1 | 1 |

图 2.65　RS 触发器的 δ 周期

δ 可以看作一个非常小的时间单位，但它在现实中又是存在的，它保证了仿真时的因果关系，从而使仿真结果不依赖于模型中各部分的仿真执行顺序。这一特性依赖于新的信号值的计算与信号赋值的分离。在一个含如下语句的模型中：

```
a < = b;
b < = a;
```

信号 a 与信号 b 的值总会被互换，如果赋值是即时发生的，则最终的结果将依赖于执行赋值操作的顺序。因此，VHDL 模型具有确定性。这也就是从一个具有确定行为的实际电路的仿真中期望得到的。

在从当前时间 $T_c$ 开始执行之前，可以有随机个 δ。这时有可能会产生无限循环，这种可能性就会引起混淆。为了避免这种可能性，方法之一就是禁止使用零延时，在触发器的模型中就使用了这种方法。

使用信号来传输数值的改变，可以很容易地使用观察者模型来实现。相对于 SDF，观察者的数量是可以改变的，这取决于等待着一个信号改变的进程数量。

VHDL模型是如何进行通信的呢？VHDL的语法描述高度依赖着惟一的（Single）、集中的（Centralized）后续事件队列，在队列中存储了所有信号的后续值。这个队列的目的并不在于实现异步的消息传递。因此，这一队列应该由仿真的内核来进行访问，以非分布式模式，每次只访问队列中的一个实体。然而，对于这种分布式的VHDL仿真，其效率也必然比较低。即使不使用基于消息的通信，所有模型中的组件都可以访问信号的值以及在其访问范围内的变量。因此，更多地将VHDL与基于共享内存的通信实现联系在一起。但是，基于FIFO的消息传递也可以在VHDL仿真器顶层得到实现。

#### 2.7.1.5　多值逻辑与IEEE 1164

在本书中，约定嵌入式系统都是基于二进制逻辑实现的，然而有时也需要使用两个以上的值来对某些系统进行建模。举个例子，也许系统中的电信号有不同的强度，有必要计算出当把两个或多个电信号的源连接在一起时的信号强度与电平。接下来，因此有必要区分信号（Signal）的电平（Level）与强度（Strength）：电平是对信号电压的抽象，而强度是对电压源阻抗（电阻）的抽象。使用离散信号的值来表示信号的电平与强度。**使用离散信号的强度，可以避免去求解基尔霍夫公式（Kirchhoff's Equations），同时也避免了使用电子工程领域的模拟电路模型。**对于未知的电子信号，将为其指定信号值。

在实践中，电子设计系统可以使用多值。一些系统只允许两个值，某些系统也允许9个或46个。使用这种多值的总体目的，首先是为了避免对电路网络公式的解析（如基尔霍夫定律），其次是保证当前的系统有足够的精度。接下来将展示确定多值化系统的技术以及这些多值之间的关系，同时使用电子信号的强度作为区分这些多值的关键参数。Hayes [Hayes, 1982] 对这种建立多值的系统化方法进行了讲述，被称为CSA理论。CSA是“连接器（Connector）、交换器（Switch）、衰减器（Attenuator）”的缩写，这3种模块是CSA理论的关键元素。下面将展示在大部分场景中基于VHDL模型如何使用多值。

1. 信号强度（两个逻辑值）

这是一种最简单的情况，从两个逻辑值开始，即'0'和'1'。这两个逻辑值有着相同的信号强度，即如果线上连接着'0'与'1'，则无法确定最终的信号电平值。

如果既没有将'0'和'1'信号连接到一起的情况，也不会有不同的信号强度在一个特定的电路节点交汇的情况，则两个逻辑值就足够了。

2. 信号强度（3或4个逻辑值）

在某些电路中，可能会存在一些信号不被任何输出驱动的情况。例如，当一些信号没有连接到地，也没有连接到电源或其他电路节点时。

如系统可能包含开集电极电路输出（见图2.66左图）。如果“下拉（pull-down）”晶体管PD处于关断状态，则输出也将是断开的。对于三态输出（见图

2.66右图），使能信号为'0'时，与门（图示为&）的输出也将为'0'，同时两个晶体管都将关断，从而输出A也将断开[⊖]。因此，使用适当的输入信号，输出可以被有效断开。

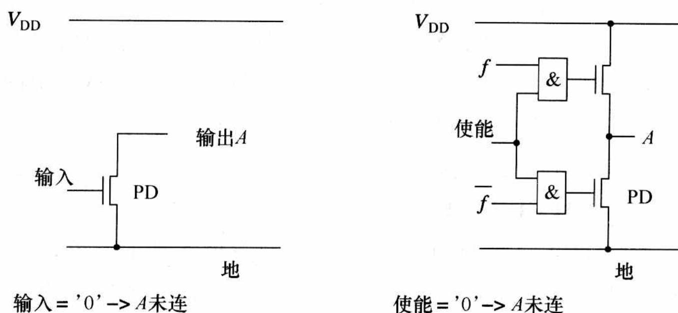



图2.66　有效的断开输出信号

很明显，在被断开时的输出信号的强度是已知最弱的。在某些情况下，信号强度'Z'比'0'和'1'还要小。因此，这样的输出信号，其电平是未知的。这种信号强度被称为'Z'态。如果一个值为'Z'的信号与另一个信号相连接，则结果信号的电平完全取决于另一个信号。举例来说，如果两个三态输出被连接到同一总线上，其中一个为'Z'，则总线电平将取决于第二个信号的值（见图2.67）。

在VHDL中，每个输出信号都与一个驱动信号相关：从多个驱动信号计算出的同一个信号称为**真值**（Resolution），用于计算此真值的函数被称为**真值函数**（Resolution Functions）。

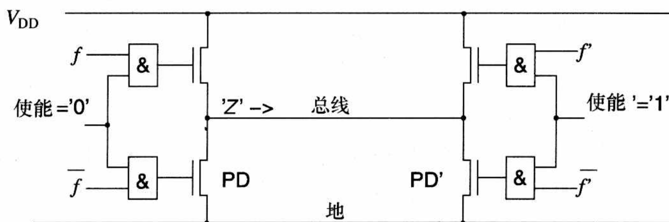



图2.67　总线受右侧输出控制

在许多场景中，逻辑值集合{'0','1','Z'}被扩展为{'0','1','Z','X'}。'X'代表了一个信号强度与'0'和'1'一样，但信号电平未知的信号。准确地说，使用'X'来表示可能为'0'或'1'的未知信号值，或者某些既不是'0'

⊖　在实践中，上拉（pull-up）晶体管一般是耗尽型晶体管，其三态输出需要被反转。

也不是'1' 的信号值[⊖]。

即使有多个驱动源连接在一起，如果清楚'0'、'1'、'$Z$' 和'$X$' 之间的转换顺序，仍然可以比较容易地对真值进行计算。图 2.68 以哈斯图（Hasse Diagram）来展示这 4 个值之间的转换顺序。

此图中的有向边反映了信号值之间的控制与转换关系。有向边定义的关系可以表示为“>”符号。如果 $a > b$，则 $a$ 控制 $b$：'0' 与'1' 控制'$Z$'，而'$X$' 控制其他的值。基于“>”所表示的关系，定义另一种关系“≥”：即当“$a \geq b$”时，含有“$a > b$”或“$a = b$”。

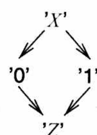


图 2.68 {'0','1','$Z$','$X$'} 的转换顺序

定义对两个信号的 sup 运算，它将返回两个信号值中的最小值（Supremum）。对于 $c \geq a$ 与 $c \geq b$，$c$ 是 $a$ 与 $b$ 之中的最小值。如 sup（'$Z$','0'）='0'、sup（'$Z$','1'）='1' 等。真值函数将根据上面的定义来计算 sup。此处的最小值与 CSA 理论中的connect 元素相对应。

3. 信号强度（7 种信号值）

在许多电路中，两种信号强度并不能满足对电路的描述需要。使用耗尽型晶体管是需要多种信号值的常见情况（见图 2.69）。

耗尽型晶体管的作用与通过向电源电压 $V_{DD}$ 提供低阻抗通路的电阻类似。前面所述的“下拉型晶体管”PD 与耗尽型晶体管，都是电路中节点 $A$ 的驱动源，节点 $A$ 的信号值可以使用真值函数来计算。下拉晶体管 PD 的驱动值可能为'0' 或'$Z$'，这取决于 PD 的输入。耗尽型晶体管的信号值比'0' 和'1' 弱，它与'1' 的信号电平一样。使用'$H$' 来表示耗尽型晶体管对节点 $A$ 的驱动值，称其为“1 弱逻辑”，相似地，可以使用'$L$' 来表示 0 弱逻辑。'$H$' 与'$L$' 的可能连接产生的结果称为“未定义弱逻辑”，用'$W$' 来表示。这样，就有了 3 种信号强度以及 7 种逻辑值 {'0','1','$L$','$H$','$W$','$X$','$Z$'}。真值同样可以使用这 7 个值之间的偏序关系来进行计算，相应的偏序关系如图 2.70 所示。

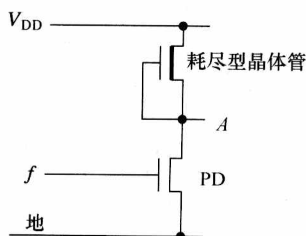


图 2.69　耗尽型晶体管的输出

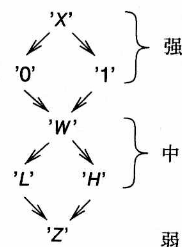


图 2.70　集合 {'0','1','$L$','$H$','$W$','$X$','$Z$'} 的偏序关系

⊖ 还有许多对'$X$' 的其他解释，但在本书中此处阐述的是最有用的说法之一。

这种偏序关系同样可以用于定义 sup 操作，它可以返回两个参数中最弱的值，如 sup（'*H*','0'）= '0'，sup（'*H*','*Z*'）= '*H*'，sup（'*H*','*L*'）= '*W*'。

'0' 与'*L*' 表达的信号电平值一样，但信号强度不一样，'1' 与'*H*' 也与此类似。增强信号强度的器件称为**放大器**（Amplifiers），减弱信号强度的器件称为**衰减器**（Attenuators）。

4. 10 个信号值（4 种信号强度）

在某些场景下，3 个信号强度并不够。例如，在导线上可存储电荷的电路。导线的电平在电路的中间过程中被充电为相应的'0' 或'1'。这些存储的电荷可以控制某些（高阻抗）晶体管的输入。但是，即使这些导线与最弱的信号源（'*Z*' 除外）相连，它们也将无法存储并保持信号值。

例如在图 2.71 中，使用专门的输出来驱动总线。

图 2.71 中的总线有一较高的容性负载 *C*。当函数 *f* 为 0 时，将 $\phi$ 设置为'1' 从而对电容 *C* 充电；而后将 $\phi$ 设置为'0'。当函数 *f* 的值变为'1' 时，停止对总线充电。由于耗尽型晶体管的阻抗较大，使用输出对总线充电的过程会比较缓慢，如图 2.69 所示，这也就是在电路中使用预充电的关键原因。通过常规的下拉晶体管 PD 放电的过程则比较迅速。

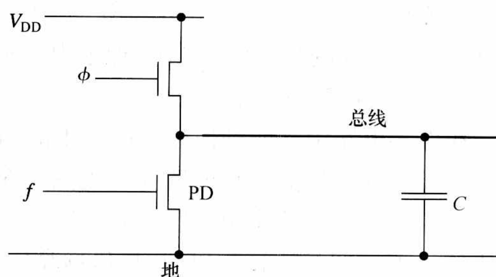



图 2.71　总线预充电

对于这样的模型，还需要比'*H*' 和'*L*' 更弱，但又比 '*Z*' 更强的信号，称这样的值为“甚弱信号值”，并且使用'*h*' 与'*l*' 来表示，甚弱信号值中的未知值则用'*w*' 来表示。这样，就得到了 10 个值｛'0','1','*L*','*H*','*l*','*h*','*X*','*W*','*w*','*Z*'｝。使用这些值，也可以得到它们之间的偏序关系（见图 2.72）。

5. 5 个信号强度

到目前为止，均没有讨论过电源信号。它们比目前讨论的最强信号还要强。如果将电压信号也考虑在内，则可以形成 46 值的集合［Coelho，1989］，但通常都很少使用这种模型。

6. IEEE 1164

在 VHDL 中，除了支持基本的二值逻辑外，并没有预先定义信号值的数量。但是可以在 VHDL 中自行定义，不同的 VHDL 模型可以使用不同的数值集。

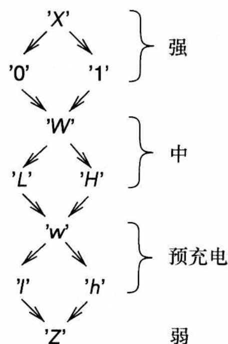



图 2.72　｛'0','1','*L*','*H*','*l*','*h*','*X*','*W*','*w*','*Z*'｝的偏序关系

但是，如果以这样的方式来提升 VHDL 能力，会给模型的可移植性带来严峻考验。为了简化 VHDL 模型的移植，IEEE 定义了一组标准的数值集。这一标准被称为 IEEE 1164，并且在许多系统模型中被使用。IEEE 1164 有 9 个值：{'0','1','L','H','X','W','Z','U','-'}，这其中的前 7 个值与前面描述的 7 个信号值一致。'U' 表示未初始化的值，它被仿真器用于表示未明确初始化的信号。

'-' 表示输入无关（Input Don't Care），需要对这个值作一些解释：通常硬件描述语言都被用于描述布尔函数。VHDL 的select 语句是实现这一功能的简单方式，它与其他语言中的switch 与case 语句类似，但它又与 ADA 中的select 语句不一样。

例如，假定要表达如下的布尔函数：

$$f(a, b, c) = a\overline{b} + bc$$

假定 $a = b = c$ ='0' 时 $f$ 的值未定义，如下是对此函数进行描述的简化方式：

```
f <= select a & b & c  -- &表示与
      '1' when "10-"   -- 相应于第一种情况
      '1' when "-11"   -- 相应于第二种情况
      'X' when "000"
```

如果按这种方式，如上函数可以很容易地转化为 VHDL 程序。但不幸的是，VHDL 中 select 语句所表示的内容与此处完全不一样。因为 IEEE 1164 只定义了可能的信号值的集合，它并未对'-' 作出明确定义。对上面所述的 select 语句，VHDL 工具检查选择表达式（本例中的 a & b & c）是否与 when 语句中的值相等。例如，它检查 a & b & c 是否与“10-”相等时，这其中的'-' 可以是任意值。那么，VHDL 系统将检查 c 是否等于'-'，由于任何变量均没有被赋值为'-'，因此比较结果将永远为假。出于 VHDL 自身定义的数值集的灵活性，才带来了这种无关值的不方便㊀。

2.7.1.5 节前面部分讨论的性质如下：它使得得出 IEEE 1164 建模能力的总结。IEEE 标准基于前面 3. 描述的 7 值数值集，因此它可以对包含耗尽型晶体管的电路进行建模，但它并不适合对充电设备进行建模㊁。

## 2.7.2 SystemC

由于偏向于使用软件来实现系统功能这一趋势的发展，越来越多的嵌入式系统既包含软件又包含硬件。大部分的嵌入式系统软件都是用 C 语言来实现的，例如用于实现 MPEG 1/2/4 标准的嵌入式系统，以及移动通信的 GSM 与 UMTS 编码器

㊀ 这一问题已经在 VHDL 2006 中修正 [Lewis et al., 2007]。

㊁ 如果在模型中并不需要耗尽型晶体管或者上拉电阻，可以将弱信号值理解为充电电荷。但由于上拉电阻在系统中已经非常常见，因此这种做法并不现实。

标准。这些标准通常都可以在“参考实现”中查找到，一般都是未经优化的C语言实现，不过都提供了所要求的功能。这就形成了基于VHDL与Verilog设计方法的缺点，即为了产生相应的硬件电路，这些标准都需要被重写。

硬件与软件协同仿真需要硬件与软件之间的交互。这通常会影响仿真效率，并且会带来用户接口的不一致，同时这也就要求开发者去学习多种语言。

因此，有必要对使用软件语言来表达硬件结构的技术进行研究。使用软件语言对硬件进行建模，必须要解决如下一些基本问题：

1）并发性（Concurrency），这是硬件的特点，需要在软件模型中实现；

2）能表达仿真时间（Time）；

3）能方便地表达**多值逻辑**（Multiple-valued Logic）与**真值**（Resolution）；

4）需要保证大部分功能电路的**确定性行为**（Determinate Behavior）。

SystemC™［SystemC，2010］，［Open SystemC Initiative，2005］被设计用于解决以上这些问题，它是C++的类库。SystemC可以使用C或C++来描述规范，构造出与类库之间适当的参考。

SystemC包含着进程并行执行的概念。它的仿真语法与VHDL类似，也包含了如何表达$\delta$周期。这些进程的执行是通过敏感信号表以及调用wait原语来控制的。敏感信号表的概念还包含了动态敏感信号表。

SystemC还包含着时间模型。早期的SystemC 1.0使用浮点数来表示时间，而在当前的版本标准中，更推荐使用整形数表示的时间。SystemC支持物理时间单位，如ps、ns及μs等。

SystemC的数据类型包含了所有常见的硬件种类：四值逻辑（'0'，'1'，'X'以及'Z'）以及不同长度的位向量。由于使用了定点数据类型，数字信号处理应用的编写得以简化。

除非使用特定的建模类型，通常确定性的行为都是不被保证的。使用命令行选项，仿真器可以以不同的顺序运行进程，这样用户可以检查仿真结果是否与进程的执行顺序有依赖关系。但对于真实场景中的复杂模型，只能对已知的非确定性行为进行建模。

基于通信与计算的分离，在不同场合的硬件重用实现得以简化。SystemC提供了通道、端口以及接口作为通信的抽象组件。这种机制被称为事务级建模（Transaction-level Modeling），由Grötker等人［Grötker et al.，2002］定义。

**定义**：“事务级建模是对从功能单元或通信架构的实现细节中分离出的模块之间的通信细节数字系统的高层模型化方法。类似总线或FIFO这样的通信机制被模型化为通道，模块使用SystemC接口类来使用这些通道。通道模型对低层的信息交换细节进行了封装，事务请求需要调用通道的接口函数。在事务级更注重数据传递的功能：传输的是何种数据、它的起点与终点在哪里。事务级较少关注数据传输的具体实现，即数据使用何种协议进行传输。这样的机制使系统级的实现更加容易，例

如，无需对连接的各种不同总线架构模型（均支持通用抽象接口）进行重新编码，使用通用的接口即可实现模块之间的交互。”

SystemC 有可能取代当前基于 VHDL 的设计流程，现在已经可以复用 SystemC 来进行硬件综合［Herrera et al.，2003a］，［Herrera et al.，2003b］，也已经有了商业化的产品。这方面的方法与应用有相关专题进行讨论［Müller et al.，2003］。SystemC 已经被制定为 IEEE 1666—2005 标准［Open SystemC Initiative，2005］。

### 2.7.3 Verilog 与 SystemVerilog

Verilog 是另一种硬件描述语言。在最初，它只被少数人使用，而后它被规定为 IEEE 1364 标准，即 IEEE 1364—1995（1.0 版本）与 IEEE 1364—2001（2.0 版本）。Verilog 中的许多特性与 VHDL 非常相似：整个设计是用相互连接的实体结构来描述的，实体的行为也可以被进一步描述，进程用于描述硬件模块的并发性。与 VHDL 一样，Verilog 也支持位向量（bitvectors）与时间单位。在某些领域，Verilog 并不如 VHDL 灵活，也缺少许多内嵌的特性。例如，标准的 Verilog 并不支持在 IEEE 1164 标准中定义的枚举类型。标准的 IEEE 1364 支持 8 种信号强度的多值逻辑，而 Verilog 支持 4 种，VHDL 对多值逻辑这一特性的支持比 Verilog 要好得多。Verilog 也有许多对晶体管级描述的特性，但 VHDL 更加灵活。例如，VHDL 可以允许将硬件实体例化为循环，这就可以用于某些结构化的描述，如可以在不手动指定 $n$ 个加法器及其连接关系的前提下来定义一个 $n$bit 的加法器。

Verilog 与 VHDL 的使用人数相当，但 VHDL 在欧洲使用得更加广泛，Verilog 在美国的使用人数则更多一些。

Verilog 的 3.0 版本与 3.1 版本即人们所知的 System Verilog，它包含着大量对 Verilog 2.0 的扩展，这些扩展包括［Accellera Inc.，2003］，［Sutherland，2003］：

1）为行为建模新增了语言元素；

2）C 数据类型，如 int、typedef 与 struct；

3）将硬件模块之间的连接重新定义为接口；

4）使用标准方法来调用 C/C++ 的函数，在某些情况下也可以在 C 中调用内嵌的 Verilog 函数；

5）在描述电路设计（Circuit Under Design，CUD）的测试环境（称为 Testbench），以及使用 Testbench 来对 CUD 进行验证的仿真方面有重大特性改进；

6）在 Testbench 中使用的来自面向对象的类；

7）创建动态进程；

8）标准的进程间通信与同步，包括信号量；

9）自动内存分配与释放；

10）为常规验证的标准化接口新增的语言特性。

由于 C 与 C++的交互能力，Verilog 也可以很容易地与 SystemC 进行交互。这些针对仿真、常规设计的验证以及与 SystemC 交互特性的增强，使 Verilog 更容易为人接受。在最近，Verilog 与 SystemVerilog 已经合并成一个标准，即 IEEE 1800—2009［IEEE，2009］。

### 2.7.4 SpecC

SpecC 是基于在嵌入式系统建模中计算与通信严格分离的一种系统级描述语言［Gajski et al.，2000］。这种分离是组件在不同上下文中可重用的前提，也使系统组件的即插即用（plug-and-play）成为可能。SpecC 将系统模型化为层次网络，网络之间通过通道来完成通信行为。SpecC 的描述符包含行为（Behaviors）、通道（Channels）与接口（Interfaces）。行为包含了端口、实例化的本地组件、私有变量与函数以及一个公共 main 函数。通道是对通信功能的封装，包含着用于定义通信协议的变量与函数。接口用于链接行为与通道，用于声明在通道中被定义的通信协议。

SpecC 也可以对嵌套的行为进行建模。图 2.73 中［Gajski et al.，2000］，组件 B 就包含了两个子组件 b1 与 b2。

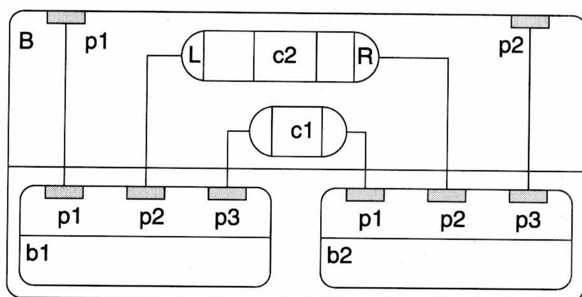



图 2.73　SpecC 结构化例子

图 2.73 中的子组件 b1 与 b2 之间通过整数 c1 以及通道 c2 进行通信。b1 与 b2 是 SpecC 结构化层次网络中的叶节点，使用关键字**par**，b1 与 b2 同时执行。图 2.73 所示的结构化层次网络可以使用如下的 SpecC 模型来描述：

```
interface L {void Write(int x); };
interface R {int Read(void); };
channel C implements L,R
  {int Data; bool Valid;
  void Write(int x) {Data=x; Valid=true;}
  int Read (void)
    {while (!Valid) waitfor (10); return (Data);} }
```

```
behavior B1(in int p1, L p2, in int p3)
  {void main (void) {/* ...*/ p2.Write(p1);} };
behavior B2 (out int p1, R p2, out int p3)
  {void main(void) {/*...*/ p3=p2.Read(); } };
behavior B(in int p1, out int p2)
 {int c1;  C c2;  B1 b1(p1, c2, c1);  B2 b2(c1, c2, p2);
 void main (void)
  {par {b1.main(); b2.main();}}
 };
```

在通道 C 所使用的接口协议中包含了读、写操作的方法，可以在不改变 B1 与 B2 行为的情况下改变这些方法。例如，通信可以是位串行（bit-serial）或并行（Parallel），都不会影响到 B1 与 B2。这一特性对于 IP 的可重用性非常重要。

在 SpecC 中，为了简化同时包含软件与硬件组件的设计，它的语法是基于 C 与 C++的。事实上，SpecC 模型都需要先转化为 C++才能进行仿真。

在系统规范阶段，SpecC 通常使用消息传递来进行通信，它基本上可以对任何种类的通信进行建模，但其仿真的实现一般需要基于非分布式的系统。SpecC 的通信模型对 SystemC 2.0 中的通信设计起到了极大的借鉴作用。

## 2.8 冯·诺依曼语言

顺序执行是冯·诺依曼语言的共同特征。同时，这类语言还允许几乎无限制地对全局变量进行访问。使用 CFSM 与计算图（Computational Graphs）的模型化设计非常适合嵌入式系统的设计。但是，标准冯·诺依曼语言的使用仍然很广泛，因此不能忽略这种语言。

KPN 与加以适当约束的冯·诺依曼语言之间的差异并不明显。对于 KPN 的每个节点，仍然有不少顺序执行的代码。对于 KPN，其建模的重点在节点内部的通信与执行细节上是不相关的，因此仍然保留着 KPN 与冯·诺依曼语言之间的差异。对于这里讲述的头两种语言，语言中即内建了通信机制；对于其他语言，主要集中在其通信机制上，并且可以选择不同的库来替换通信机制。

### 2.8.1 CSP

顺序通信进程（CSP，Communicating Sequential Processes）[Hoare，1985] 是最早包含进程间通信机制的语言之一，其通信是基于通道的。

例子：

```
process A                    process B
.....                        ......
var a ...                    var b ...
 a := 3;                      ...
 c!a; -- 输出至通道C            c?b; --从通道C输入
end;                         end;
```

两个进程都将等待其他进程到达输入或输出状态，这可以看作基于 rendezvous、阻塞（Blocking）或同步消息传递（Synchronous Message Passing）。

由于 CPS 依赖于等待来自一个特定通道输入的保证，如 Kahn 进程网络（Kahn Process Networks），因此它是确实性的。

CPS 为 OCCAM 语言奠定了基础，OCCAM 语言被作为Transputer［Thiébaut，1995］的编程语言。在 XS1 处理器［XMOS Ltd.，2010］的设计中，这种强调通信通道的方式又被重拾。

### 2.8.2 ADA

在 20 世纪 80 年代，美国国防部意识到如果不对军用设备中的软件进行强制约束，它们的可靠性与可维护性将很快成为一个巨大的问题。因此，他们决定所有的软件都必须使用同一种实时语言来进行编写，也就需要规划这样一种语言。

当时并没有能满足国防部需要的这样一种语言，因此他们决定重新进行开发。这种语言最终基于 PASCAL 语言进行编写，被称为 ADA（以 Ada Lovelace 命名，她被认为是第一位女性程序员）。ADA’95［Kempe，1995］，［Burns and Wellings，2001］是原始标准的面向对象扩展。

ADA 的一个有趣特性是它允许在进程（在 ADA 中被称为任务）中嵌套地进行声明。只要控制传递到了任务的作用域，则它们就立即开始执行。

如下例所示（根据 Burns 等人的研究［Burns and Wellings，1990］）：

```
procedure example1 is
    task a;
    task b;
    task body a is
      -- a的局部声明
      begin
        -- a的状态
      end a;
    task body b is
      -- b的局部声明
      begin
```

```
      -- b的状态
    end b;
  begin
   -- 任务a与b将在example1的第1个状态之后开始运行
   -- example1的状态
  end;
```

ADA 中的通信是另一个重要的概念。它基于rendez-vous 范例（rendez-vous Paradigm）。无论何时，只要两个任务想交换信息，先到达"会合点（Meeting Point）"的任务必须等待另一个任务也到达相应的会合点。按照 ADA 的语言，程序（Procedures）用于描述通信。在任务中被调用的程序必须使用关键字**entry** 进行声明。

例如［Burns and Wellings，1990］：

```
task screen_out is
  entry call (val : character; x, y : integer);
end screen_out;
```

在任务 screen_out 中包含着一个名为 call 的程序，它可以在其他进程中被调用。其他任务也可以通过在程序前加上任务名前缀来调用此程序：

```
screen_out.call('Z',10,20);
```

调用任务需要等待任务到达一个控制点，在控制点它才接收其他任务的调用。这个控制点使用了关键字**accept**：

```
task body screen_out is
  ...
  begin
    ...
    accept call (val : character; x, y : integer) do
    ...
    end call;
  ...
  end screen_out;
```

显然，任务 screen_out 可能会在同一时间等待多次调用。ADA 的 select 状态提供了解决这种情形的功能。

例如：

```
task screen_output is
  entry call_ch(val:character; x, y: integer);
  entry call_int(z, x, y: integer);
```

```
end screen_out;
task body screen_output is
  ...
  select
   accept call_ch ...  do...
   end call_ch;
 or
   accept call_int ... do ..
   end call_int;
 end select;
 ...
```

在这个例子中，在 call_cn 或 call_int 被调用之前，任务 screen_out 都将处于等待状态。

由于可以使用 select 状态，ADA 并不是确定性的。ADA 曾经一度在西方国家作为军事设备上使用的首选编程语言。关于 ADA 的最新信息可以从相关网站上进行查找（如［Kempe Software Capital Enterprises（KSCE），2010］）。

### 2.8.3　Java

对于 Java 而言，其通信可以通过加载不同的软件库来实现，它的计算是严格顺序执行的。

Java 是一种与运行平台无关的语言。只要机器上有 Java 解释器可以将程序编译生成的内部字节码（Byte-code）进行解释，则 Java 程序就可以运行。字节码是一种紧凑型的表达，与标准的二进制机器码相比，它需要更少的内存存储空间。显然，对于内存空间有限的片上系统应用而言，Java 的这种特性是一个优点。

同时，Java 也是一种安全机制较高的语言。C 或者 C++ 语言（如使用指针的算法）中一些潜在的危险特性在 Java 中都不存在。因此，Java 满足了嵌入式系统语言中对安全性的要求。Java 也支持异常处理，简化了运行时的错误恢复过程。由于 Java 提供了自动内存垃圾回收的机制，因此它不会因为地址空间的重新分配而导致内存泄露。这种特性避免了某些必须不间断的运行数月甚至数年的应用系统的一些潜在问题。由于 Java 包含着线程（轻量级的进程），它同时也能满足并发性的要求。

此外，由于 Java 支持面向对象的特性，它的开发系统提供了许多功能非常强大的库，因此 Java 应用能够快速得到实现。

但是，标准的 Java 并不是针对实时嵌入式系统而设计的，这也使得它缺少了许多实时嵌入式系统必需的特性：

1）Java 运行库（Run-time Libraries）的大小也会加到最终的应用程序大小之

中，而运行库可能会非常大。因此，只有那些大规模的应用才会从应用本身的紧凑型规划中获益。

2）许多嵌入式应用都需要直接控制某些 I/O 设备。但在 Java 中出于安全性的考虑，它并不允许对 I/O 设备直接控制。

3）Java 的自动垃圾回收也需要处理时间。在标准 Java 中，何时进行垃圾回收是不能预测的。因此，也就很难估计最差执行时间，而只能对运行时的状况作一个保守估计。

4）如果多个线程都准备运行，Java 不能指定各个线程之间的执行顺序。因此，Java 环境下的最差执行时间更加难以估算。

5）相对于 C 语言，Java 的效率更差。因此对于系统资源受限的系统，并不推荐使用 Java。

Nilsen [Nilsen，1998] 给出了解决这些问题的方案，如硬件支持的垃圾收集、替换 Java 中的运行调度器以及在某些内存段中加上特定标签等。

当前（2010 年）流行的 Java 编程平台有 Java 企业版（J2EE，Java Enterprise Edition）、Java 标准版（J2SE，Java Standard Edition）、Java 微缩版（J2ME，Java Micro Edition）以及 CardJava [Sun，2010]。CardJava 是原 Java 的一个子集，它主要应用在强调安全性的智能卡领域。J2ME 是智能卡以外所有嵌入式系统的 Java 开发平台。在 J2ME 中使用两个配置库：CDC 与 CLDC。其中，CLDC 主要用于移动电话，它使用 MIDP 1.0/2.0 作为其应用程序编程接口（Application Programming Interface，API）。CDC 一般用于电视机以及功能更加强大的手机。当前关于 Java 实时性编程的著作作者主要有 Wellings [Wellings，2004]，Dibble [Dibble，2008] 和 Bruno [Bruno and Bollella，2009] 等，同时还有 [Java Community Process，2002] 与 [Anonymous，2010b] 等网站。

### 2.8.4 Pearl 与 Chill

Pearl [Deutsches Institut für Normung，1997] 主要是为工业控制应用而设计的。针对控制过程与时间参考，它包含了大量特别的语言元素。Pearl 需要一个基本的实时操作系统作为开发基础。Pearl 在欧洲的许多工业控制项目中都得到了很广泛的应用。基于共享内存，Pearl 支持使用信号量进行项目级的通信。

Chill [Winkler，2002] 主要用于电话交换站，它是 CCITT 的标准之一，在电信设备上得到了比较广泛的应用。Chill 是 PASCAL 的一种扩展。

### 2.8.5 通信库

标准的冯·诺依曼语言并没有内嵌通信原语，但是它允许以函数库的形式添加通信功能。这些函数库在支持局部系统通信的同时，也开始支持长距离通信。网络协议就是函数库中的例子，它已经得到了越来越广泛的使用。在本书讲述系统软件

的章节对函数库还作了进一步的介绍。

## 2.9 硬件建模的层次

在实践中，工程师会从多种不同的抽象层次来开始一个设计周期。在某些场景，首先考虑的是系统整体行为的高层描述；但在另一些场景，设计却是从对底层电路规范的抽象开始。对于设计中的每一个层次，都会有大量的建模语言可供选择。在下面的部分，将讨论一系列可能的建模层次。出于本书内容的考虑，对一些较底层的描述也安排在了此处，系统的设计不应该从这些底层着手。下面是一些常用建模层次的名称及特征：

1）**系统级模型**（System Level Models）：系统级这一术语的定义并不是非常清晰。在此处它表示的是整个嵌入式系统以及内嵌了信息处理单元的系统（“产品”），同时它可能还包含环境特征（系统的物理量输入，如轨道、天气情况等）。很显然，对于这样包含着机械结构与信息处理的系统级模型，很难找到合适的仿真环境。可能解决这一问题的方法有 VHDL-AMS（VHDL 向模拟部分的扩展）、SystemC 或者 MATLAB。MATLAB 与 VHDL-AMS 支持对偏微分方程的建模，这是对机械系统建模的一个关键特性。这些仿真模型同样也可以用于嵌入式系统的综合，它还可以对系统的信息处理部分进行建模，但这已经是比较有挑战性的事情了。如果做不到这一点，则可能需要在不同的模型之间进行转换，而这种转换过程非常容易出错。

2）**算法级**（Algorithmic Level）：在这一层级，需要对将在嵌入式系统中使用的算法进行仿真。例如，为了评估最终的视频质量，可能需要提前对 MPEG 视频编码算法进行仿真。对于算法仿真，它与最终使用的处理器以及指令集无关。

仿真时使用的数据类型也许会比最终使用的数据类型有更高的精度。例如，仿真时的 MPEG 使用双精度浮点型，而最终的嵌入式系统很难也使用这样的精度。如果在仿真时使用的 bit 信息能与最终实现时的 bit 信息完全对应，这样的模型就被称为位真（bit-true）模型。将非位真模型转换为位真模型需要工具软件的支持。

这一层级的模型可能会包含着单个处理器或一系列的协处理器。

3）**指令集级**（Instruction Set Level）：在这种情况下，算法已经被编译成了所使用的处理器的指令。在这一层级的仿真可以计算出所执行的指令数量。指令集级有许多种变异：

① 在简化的模型中，通常只对指令进行功能仿真而并不考虑时序。在汇编参考手册［指令集架构（Iinstruction Set Architecture，ISA）］的信息已经足够用于定义此类模型。

② **事务级建模**：在事务级模型中，诸如总线读写、不同模块之间的通信等事务都被模型化，它比 cycle-true 模型（参考下文）包含的细节要少，它更强调仿真

速度［Clouard et al.，2003］。

③ 在更精细的模型中，可以使用cycle-true 指令集仿真。在这种场景下，可以计算出运行一个应用需要的精确的时钟周期数。定义一个 cycle-true 模型需要关于处理器硬件的详细信息从而对模型进行校正，如流水线阻塞、资源限制以及内存的等待延时等。

4）**寄存器传输级**（Register-Transfer Level，RTL）：在这一层级，在寄存器传输级对所有模块进行建模，包含算术/逻辑单元（Arithmetic/Logic Units，ALU）、寄存器、内存、多路复用器以及解码器。这一层级的模型通常都是 cycle-true 的。根据这一层级的模型进行自动综合并不是很大的挑战。

5）**门级模型**（Gate-level Models）：在这种场景下，模型以逻辑门作为其基本元件。门级模型描述了信号跃迁的详细信息，因此它可以用于系统的功耗估算。相比 RTL，门级模型也可以得到更为精确的时延信息。但是门级模型不包含总线长度以及容抗等信息，因此计算出的功耗与时延并不十分准确。

在只使用逻辑门来表示布尔函数的一些场景中，也常常使用“门级模型”这一术语。在这种模型中的逻辑门并不一定表示的是真实的门电路，只考虑了逻辑门的行为，并未考虑它们是否表示着真实的物理元件。更准确地讲，这样的模型应该被称为“布尔函数模型（Boolean function models）”[⊖]，但这一术语很少被使用。

6）**开关级模型**（Switch-level Models）：开关级模型使用开关（晶体管）作为其基本器件，使用数字量模型（参考 2.7.1.5 节关于数字量的描述）。与门级模型相比，开关级模型更适合描述信息的双向传递。

7）**电路级模型**（Circuit-level Models）：电路理论及其元器件（电流源、电压源、电阻、电容、电感以及半导体的宏模型）组成了这一层级的仿真基础。这里的仿真还包含着偏微分方程，只有当且仅当半导体电子的行为是线性的（或接近线性），这些公式才是线性的。这一层级下使用最多的仿真器是 SPICE［Vladimirescu，1987］及其变异方法。

8）**布局模型**（Layout Models）：布局模型反映了真实电路的布局，它包含着电路的几何（Geometric）信息。因为几何信息并不能直接提供与电路行为相关的信息，因此布局模型不能被直接仿真。可以从布局模型与较高层次的行为描述中，或从布局中对电路进行分解，又或者使用这一层级中与电路元件相关的知识来对电路行为进行推演。在典型的设计流程中，总线长度与相应的容抗都是从布局分解得知的，这些信息而后又作为**反标注**（Back-annotated）添加到较高层次的描述中。这样，关于系统时延与功耗的估算就能更加准确。

9）器件与工艺模型（Process and Device Models）：在更低的层级上，还可以

⊖ 这些模型可以使用二元决定图（Binary Decision Diagrams，BDD）来表示［Wegener，2000］。

对工艺流程进行仿真。使用这些模型信息，还可以计算出器件（晶体管）参数（增益、容抗等）。

## 2.10　计算模型的比较

### 2.10.1　比较的标准

计算模型可以通过许多标准进行比较。如 Stuijk [Stuijk，2007] 与 MoC 可以按如下标准进行比较：

1）**易表达性**（Expressiveness）与**简洁性**（Succinctness）分别说明了哪类系统可以被模型化以及它能达到怎样的紧凑度。

2）与调度算法的可用性以及运行时支持的需求相关的**可分析性**（Analyzability）。

3）**实现效率**（Implementation Efficiency）受所需要的调度策略以及代码大小的影响。

图 2.74 根据以上这些标准对数据流模型进行了分类。

图 2.74 反映了 Kahn 处理网络的易表达性：Kahn 处理网络是图灵完全的（Turing-complete），也就是说任何可以使用图灵机（Turing Machine）来完成计算的问题，同样可以使用 KPN 来完成计算。图灵机通常被作为通用计算的标准模型 [Herken，1995]。但是，KPN 的终端属性与缓冲区大小上限一般不容易分析。另一方面，由于 SDF 图不能对控制流进行建模，因此它并不是图灵完全的，但死锁（Deadlock）与缓冲区大小上限却比较容易进行分析。同构的 SDF（Homogeneous SDF，HSDF）图（所有图的比率相等）更不易表达，但更容易分析。

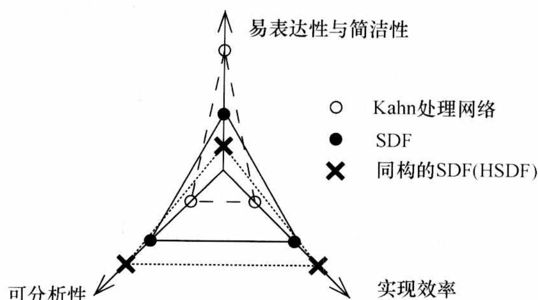


图 2.74　数据流模型的比较

可以通过所支持的进程类型来对 MoC 进行比较：

1）**进程数量**（Number of Processes）可以是**动态**（Dynamic）或**静态**（Static）的。如果每个进程都已经对应可以完成某部分硬件功能，同时也不必考虑"热插拔（hot-plugging）"（动态地改变硬件架构），则使用静态数量的进程可以简化实现。否则，系统就需要支持动态的进程创建（与结束）。

2）进程可以是**静态嵌套**（Nested）或在同一级进行声明。例如，状态图允许嵌套的进程声明，但 SDL 却不允许。嵌套这一方式主要是源于封装性方面的考虑。

3）有多种不同的**进程创建**（Process Creation）方式。可以通过在源代码中声

明进程，通常使用 fork 与 join 机制（如 UNIX 操作系统就支持此类方式），或者明确调用进程创建函数。

多种以数据流为基础的计算模型的易表达性如图 2.75 所示［Basten, 2008］。虚线表示的是一些没有在本书中讨论的模型，如 MoC。

至今描述的所有 MoC 与语言中，没有任何一种能完全满足嵌入式系统规范语言的要求。图 2.76 展示了一些语言的关键特性概要。

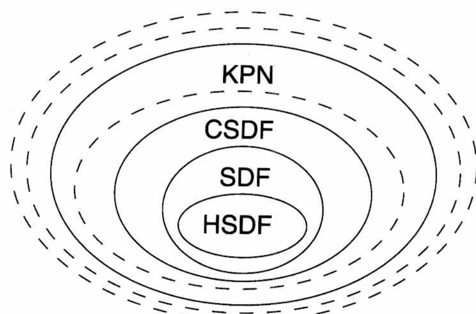



图 2.75　数据流模型的易表达性

| 语言 | 行为层次 | 结构层次 | 编程语言要素 | 支持异常 | 动态进程创建 |
|---|---|---|---|---|---|
| 状态图 | + | - | - | + | - |
| VHDL | + | + | + | - | - |
| SpecCharts | + | - | + | + | - |
| SDL | +- | +- | +- | - | + |
| Petri nets | - | - | - | - | + |
| Java | + | - | + | + | + |
| SpecC | + | + | + | + | + |
| SystemC | + | + | + | + | + |
| ADA | + | - | + | + | + |

图 2.76　语言对比

有意思的是，SpecC 与 SystemC 在图 2.76 中满足了列出的所有需求。但是事实上有许多其他需要并没有在图 2.76 中列出（如关于 deadlines 的精确划分等）。由于某些需求在本质上就是有冲突的，因此单一的 MoC 或语言基本上不可能完全满足所有需求。对于低约束的实时需求，如果使用支持硬实时要求的语言会非常不方便；同样，适合分布式控制应用领域的语言可能并不适合以分立数据流为主的应用。因此，许多时候都要折中地进行选择，或者使用多种模型。

那么，在实践中应该使用怎样的折中方案呢？在实践中，早期的嵌入式系统编程通常都使用汇编语言编程，它的程序规模一般都比较小。而后主要是使用 C 或 C 的派生语言。由于嵌入式系统软件复杂度不断增长，在 C 语言之后引入了高层语言。面向对象的语言与 SDL 都提供了下一等级的抽象，类似于 UML 的语言都需要在设计早期获取设计规范。在实践中，可以按图 2.77 使用这些语言。

根据图 2.77，类似于 SDL 或状态图的语言可以被转化为 C 语言而后编译。如果有从 SDL 或状态图到 VHDL 的转换器，则它们可以实现硬件上的功能。C 与 VHDL 作为中间语言已经被使用了很长时间。得益于向汇编语言转换的良好机制，Java 并不需要中间步骤。类似地，各种图之间的转换是非常灵活的。如 SDF 图可

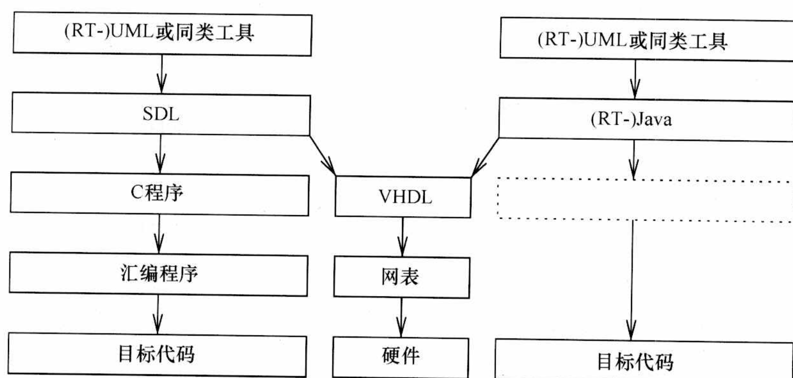



图 2.77　组合使用多种语言

以很容易地转换为 Petri 网［Stuijk，2007］的一个子类，同样，它也可以与 Karp 与 Miller［Karp and Miller，1966］发起的计算图模型（Computation Graph Model）子类相对应。在将各种计算模型联系起来的过程中，富有条理的规范与技术起到了积极的推动作用［Chen et al.，2007］。

M. Radetzki［Radetzki，2009］的著作中讲述了关于嵌入式系统设计的多种语言。Popovici 等人［Popovici et al.，2010］将 Simulink 与 SystemC 进行了组合运用。

## 2.10.2　UML

UML™包含的图可反映多种 MoC 模型。图 2.78 根据 MoC 的一些特点对当前已经提及的 UML 图进行分类。

| 通信组件 | 共享内存 | 消息传递 | |
|---|---|---|---|
| | | 同步 | 异步 |
| 未定义的组件 | 用例 | | |
| | | 顺序图，时序图 | |
| 有限状态机 | 状态图 | - | - |
| 数据流 | （不使用） | 数据流图 | |
| Petri网 | （不使用） | 活动图 | |
| 分布式事件模型 | - | - | |
| 冯·诺依曼模型 | - | - | |

图 2.78　UML 中可用的计算模型

图 2.78 展示了在早期的设计阶段，UML 如何覆盖多种计算模型。通常，对通信语义的定义都并不精确，从这个角度考虑，关于 UML 的分类也无法非常精确。基于以上提到的各种图，有以下 8 类图可用于建模：

1）部署图（Deployment Diagrams）：此类图对于嵌入式系统是非常重要的，它们描述了系统的“运行时结构”（硬件或软件节点）。

2）包图（Package Diagrams）：包图将软件分割成软件包，它们与状态机中的模块图非常类似。

3）类图（Class Diagrams）：类图描述了对象类之间的继承关系。

4）通信图（Communication Diagrams）[在UML 1.x中也被称为协作图（Collaboration Diagrams）]：通信图表达了类与类之间的关系以及它们之间交互的消息。

5）构件图（Component Diagrams）：构件图描述了在应用或系统中使用的构件。

6）对象图（Object Diagrams）、交互概图（Interaction Overview Diagrams）、复合结构图（Composite Structure Diagrams）：这3种类型的图都很少使用，它们有些其实是其他图的特定类型。

许多工具都提供了对多种不同类型图的一致性检查功能。由于UML的语义在最初就没有清晰的定义，因此要做到完整的一致性检查基本上不太可能。它是否是UML创始者的本意，这是一个有争议的话题，它也许是出于希望设计者在最初始设计阶段不要过多地考虑精确的语义的目的。因此，只有在UML与其他一些可执行语言组合时，才能从中得到精确的、可执行的规范。有一些工具已经将UML与SDL [IBM, 2009] 和C++结合了起来。事实上，有许多人也会尝试在设计初期去定义UML的语义。

1.4版本的UML并不是针对嵌入式系统而设计的，因此它缺乏许多嵌入式系统建模的特性。典型地，它缺少如下特性 [McLaughlin and Moore, 1998]：

1）从软件中划分的任务与进程不能模型化；

2）时序行为完全不可描述；

3）基本的硬件模块不可描述。

由于嵌入式系统中软件的大量增长，嵌入式系统UML的重要性也有所提高。针对UML在支持实时应用领域的扩展，许多学者都提出了建议 [McLaughlin and Moore, 1998]，[Douglass, 2000]。在UML 2.0版本中已经进行了多项扩展，它包含13种图类（UML 1.4包含9种）[Ambler, 2003]。UML的概要文件也考虑了实时系统的需求 [Martin and Müller, 2005]，[Müller, 2007]，它包含了带约束的类图、图示、符号以及一些（部分）语义。以下是一些UML概要文件 [Müller, 2007]：

1）可调度性、性能与时间规范（Schedulability, Performance and Time Specification, SPT）[Object Management Group (OMG), 2005b]；

2）测试 [Object Management Group (OMG), 2010a]；

3）服务质量（Quality of Service, QoS）与容错 [Object Management Group (OMG), 2010a]；

4）称为SysML的系统建模语言 [Object Management Group (OMG), 2008]；

5）实时嵌入式系统的建模与分析（Modeling and Analysis of Real-Time Embed-

ded Systems, MARTE）[Object Management Group（OMG）, 2009]；

6）UML 与 SystemC 的互操作性 [Riccobene et al. , 2005]；

7）SPRINT 概要文件知识产权（Intellectual Property, IP）的可重用 [Sprint Consortium, 2008]。

使用这些概要文件可以实现许多组合功能，例如将时序信息与流程图结合起来。但概要文件有可能是不兼容的。UML 是为建模而开发的一种图形化语言，它又留下了大量的开放型语义问题，这些问题使得 UML 可以在实现时进行自动综合 [Müller, 2007]。

### 2.10.3 Ptolemy Ⅱ

Ptolemy 是一种建模仿真环境，它重点用于异构系统的建模、仿真与设计。针对嵌入式系统，则主要是混合了多种技术的 MoC。例如，它可以描述模拟与数字电子、硬件与软件、电子与机械设备。Ptolemy 支持多种不同的应用领域，如信号处理、控制工程、渐进决策以及用户接口等。尤其需要注意嵌入式软件的生成，它主要是根据适合于一些特定领域的 MoC 来产生软件。Ptolemy 的第二版（Ptolemy Ⅱ）支持如下 MoC 及其相应领域：

1）通信顺序进程（Communicating Sequential Processes, CSP）。

2）连续时间（Continuous Time, CT）：这一模型适合于机械系统与模拟电路。它是通过对一系列微分方程的扩展来支持这一领域的。

3）离散事件（Discrete Event, DE）模型：许多仿真器都使用了这一模型，如 VHDL 仿真器。

4）分布式离散事件（Distributed Discrete Events, DDE）：由于后续事件采用单一队列，因此离散事件系统往往难以进行并行仿真。对这种数据结构的改变至今没有取得很好的成果，因此它主要针对一些特定（实验性）的领域。它的语义定义使其分布式仿真比在 DE 模型中有更高的效率。

5）有限状态机（Finite State Machines, FSM）。

6）进程网络（Process Networks, PN）：使用 Kahn 进程网络。

7）同步数据流（Synchronous DataFlow, SDF）。

8）同步/激励（Synchronous/Reactive, SR）MoC：这种模型使用离散时间，但不需要在每个时钟节拍都有对应值。Esterel 就属于这类建模语言。

上面的清单清晰地展示了在 Ptolemy 项目中对不同计算模型的关注点。

## 2.11 思考题

1. 列出嵌入式系统规范语言至少 6 个需求。

2. 使用 levi 仿真软件 [Sirocic and Marwedel, 2007d] 对巴黎、布鲁塞尔、阿

姆斯特丹与科隆之间运行的列车进行仿真。修改软件中的例子，从而使任意两个站点之间有两条独立的轨道，并且展示出含 10 辆列车的一种（仲裁）调度方案。

3. 假定给出了图 2.79 所示的状态图。

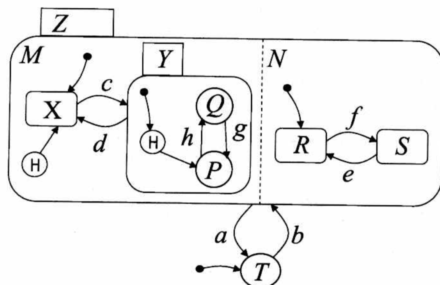



图 2.79　状态图的例子

另外，假定输入事件的顺序为：*b c f h g h e a b c*。在图 2.80 所示的表中，标识出加上特定输入后状态图所处的状态。注意，*H* 表示了历史机制。

| | *M* | *N* | *P* | *Q* | *R* | *S* | *T* | *X* | *Y* | *Z* |
|---|---|---|---|---|---|---|---|---|---|---|
| (Reset) | | | | | | | v | | | |
| *b* | | | | | | | | | | |
| *c* | | | | | | | | | | |
| *f* | | | | | | | | | | |
| *h* | | | | | | | | | | |
| *g* | | | | | | | | | | |
| *h* | | | | | | | | | | |
| *e* | | | | | | | | | | |
| *a* | | | | | | | | | | |
| *b* | | | | | | | | | | |
| *c* | | | | | | | | | | |

图 2.80　状态图的状态示例

4. 请解释：如果状态图遵从 StateMate 的语义，它是确定性的吗？

5. 本书中讨论了哪 3 种类型的 Petri 网？

6. Petri 网中的某一类允许每个库所有多个非确定性的令牌，这一类网络的数学模型中使用了哪些组件？提示：$N=(P, \cdots)$。

7. 哲学家就餐问题的紧凑模型跟什么比较类似？

8. CSA 理论会有 2、3 或 4 个逻辑强度，相应于 4、7 和 10 这些逻辑值。在 IEEE 1164 中使用了多少种强度与数值？

请以表格的形式列出 IEEE 1164 值的半序。确定 IEEE 1164 的哪些值不在此半序范围中，以及这些值的意义是什么。

9. 哪一种电路可以使用 IEEE 1164 进行建模：互补 CMOS 输出、带耗尽型晶

体管的输出、开集电极输出、三态输出、总线预充电（如果同时使用耗尽型晶体管）。

10. 假定给出图 2.81 所示的总线。包含 & 符号的矩形表示与门。

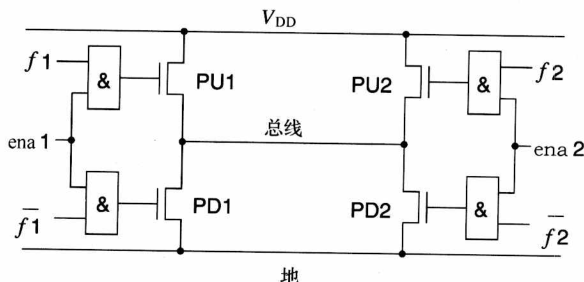



图 2.81　三态输出驱动的总线

如果两个输入使能都置为'0'（即 ena1 = ena2 = '0'），总线上将会有哪些 IEEE 1164 的值？如果 ena1 = '0'，ena2 = '1'，且$f2$ = '1'，则总线上又会有哪些 IEEE 1164 的值？

11. 使用 levi 仿真软件［Sirocic and Marwedel，2007b］，仿真一个用以计算斐波纳契数列（Fibonacci Numbers）的 Kahn 处理网络。

12. 下列哪一种语言使用异步消息传递通信机制：状态图、SDL、VHDL、CSP、Petri 网。

13. 下列哪一种语言使用广播机制来更新变量：状态图、SDL、Petri 网。

14. UML 支持下列哪种图：顺序图、记录图、Y 图、用例、活动图、电路图。

# 第3章 嵌入式系统硬件

## 3.1 简介

在嵌入式信息—物理系统的设计中，需要对硬件与软件进行综合考量，这也是它的一个重要特点。在基于平台的设计理念中，对硬件与软件模块的选择是其核心。图3.1中的设计流程信息，相应地展示了对可用硬件模块的需求，而在接下来的内容中，将对嵌入式系统硬件的本质进行阐述。

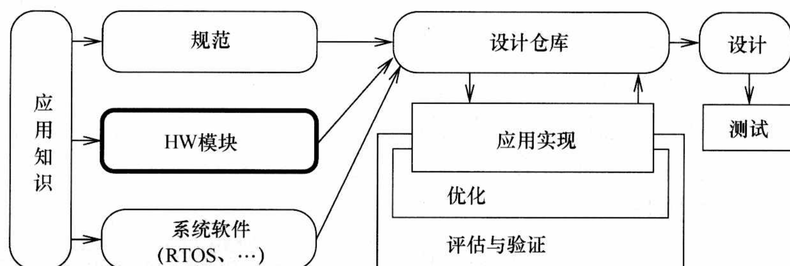



图3.1 简单的设计流程

与个人电脑中的标准化硬件不同，嵌入式系统的硬件多种多样。对于如此庞大的硬件种类，很难对所有硬件类型都进行一个全面的描述，不过将对在大部分系统中具有共性的特征模块进行概述。

在很多的信息—物理系统，尤其是控制系统中，硬件一般都工作在回路状态（见图3.2）。

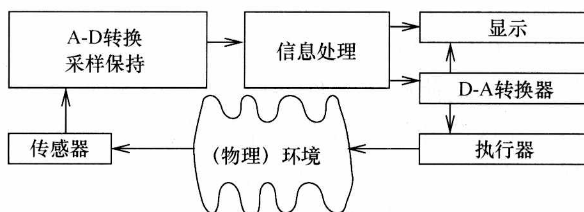



图3.2 硬件回路

在这个控制回路中，传感器首先对物理环境中的信息进行采集。通常，传感器的输出都是连续的模拟量。在本书中，规定信息都必须以离散数字量的形式进行处理。从连续模拟量到离散数字量的转换，通常由两种电路完成：采样保持电路与模-数（A-D）转换电路。经过这些转换后，信息就可以以数字量的形式进行处理了。这时的处理结果可以在显示设备上进行呈现，执行器也可以利用它们来对实际的物理量进行相应控制。大部分的执行器都是模拟的，因此有必要将数字量再转换为模拟量。

很显然，这样的模型很适合控制类的应用。对于其他一些应用，它也可以用于初期的评估参考。接下来，将根据图3.2中的结构，对信息—物理系统中硬件模块的本质特征进行描述。

## 3.2　输入

### 3.2.1　传感器

现在，将开始对传感器进行详细讨论。对于现有的物理量，基本上都有可对其进行测量的传感器，如质量、速率、加速度、电流、电压、温度等。传感器理论[Elsevier B. V. , 2010a] 描述了很多物理影响与传感器关系，这样的例子如电磁感应（在变换的电场中产生电压），光电效应等。对于某些化学反应，也存在相应的传感器 [Elsevier B. V. , 2010b]。

最近几年间，有大量新型的传感器设计产生，这其中大部分都得益于智能系统的发展，它们为现代传感器技术起到了重要的推动作用。我们的讲述不可能完全覆盖物理信息系统技术，只能列举一些关键的技术要点作为例子：

1）**加速度传感器**：图3.3展示了基于微系统技术制造的一种小型加速度传感器。传感器中心包含一个很小的质量块，当传感器加速运动时，质量块的位置发生变化，从而导致与它保持连接的导线电阻值发生变化。

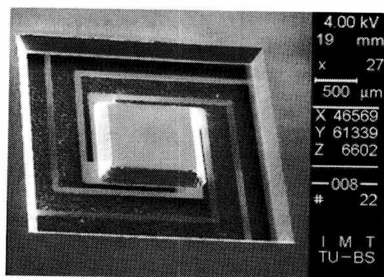



图3.3　加速度传感器（courtesy S. Bütgenbach，IMT，TU Braunschweig，© TU Braunschweig，德国）

2）**雨量传感器**：为了减少对驾驶员的干扰，一些汽车装备了雨量传感器，刮水器可以根据雨量的大小进行自动调节。

3）**图像传感器**：当前存在两种基本的图像传感器：电荷耦合器件（CCD）与CMOS传感器。在这两种传感器中，都使用了光传感器阵列。CMOS传感器阵列的架构与标准内存相似：单个像素都可以被随机寻址与读取。CMOS传感器使用CMOS的标准集成电路 [Dierickx，2000]。正是因为这样，传感器部分与逻辑电路部分才能被集成在同一块芯片上。同时有一些预处理的工作也可以在传感器芯片上完成，这被称为智能传感器。CMOS只需要单个标准电源，同时它的对外接口也非常简单，因此基于CMOS的传感器往往比较便宜。

相对于CMOS传感器，CCD传感器技术在光学特性上做了更多的优化。在CCD技术中，像素点反映了电荷的变化，所有像素的信号叠加在一起，最终形成了整幅画面。电荷连续变化时也向CCD标明了它们在面板上的位置。对于CCD传感器，接口更加复杂。

这里没有一个明确的标准来选择最合适的图像传感器。在过去几年，CMOS 传感器采集的图像质量得到了很大的提升。使用 CCD 或 CMOS，均可以得到一样的图像质量。但是，CMOS 传感器通常比 CCD 传感器的功耗要大。因此，如果是设计低功耗的系统，应当首先考虑使用 CCD 传感器。而如果更多地考虑成本因素，则可以选用 CMOS 传感器。如果是设计智能传感器，则首先选择 CMOS 型。出于对低功耗的考虑，有实景播放功能的相机，一般都使用 CMOS 传感器［Belbachir, 2010］。对于其他相机，可能需要根据实际应用来进行选择。

4）**生物传感器**：由于对移动通信及移动设备更高安全性的需求，生物认证领域保持了持续增长。基于密码验证的认证方式有很多局限性（如被盗或密码丢失），因此智能卡、生物传感器及生物认证在近年得到了较多的关注。生物认证主要是对用户的当前身份的真实性进行验证，这些验证方式一般有虹膜识别、指纹识别以及面部识别等。指纹传感器一般使用与集成电路制造一样的 CMOS 技术［Weste et al., 2000］。如在笔记本电脑上验证用户的指纹，只有验证通过的合法用户，才能使用系统［IBM, 2002］。上面描述的 CCD 与 CMOS 传感器都可以用于面部识别。在生物认证中，也存在认假率与拒真率的问题，它不可能做到像密码那样完全匹配。

5）**人工眼**：近年来，人工眼得到了诸多关注，一些项目希望它能对视觉产生影响，一些项目则希望它能提供一种间接的成像功能。

如 Dobelle 研究所将微型摄像机安置在眼镜上，摄像机收集的信号被计算机处理并转化为电子脉冲。通过与人体直接相连的电极，这些脉冲信息被直接送入大脑。大脑所感应到的分辨率大约是 128 × 128 像素，它足以使盲人可以在某些受控区域驾驶汽车［The Dobelle Institute, 2003］。

最近，将图像转化为声音的技术与有了突破，显然，它对人体具有更小的入侵性。

6）**无线射频识别（RFID）**：无线射频识别是基于智能标签对无线射频信号进行反馈［Hunt et al., 2007］。智能标签由集成电路与天线组成，它向RFID 阅读器发送其自身的身份信息。智能标签与阅读器之间的最大距离，取决于智能标签的类型。这种技术可以应用在多种目标上，人或者动物也可以被识别。

7）**其他传感器**：其他一些传感器包括：压力传感器、接近传感器、发动机控制传感器、霍尔传感器等。

传感器都会产生信号，以数学化的形式可以如下定义：

**定义**：信号$\sigma$ 是从时域 $D_T$ 到数值 $D_V$ 的映射：

$$\sigma: D_T \rightarrow D_V$$

信号在时域可以是连续或者离散的，同样它在数值上也可以是连续或者离散的。

### 3.2.2　离散系统：采样保持电路

当前所有的数字计算机都是在**离散**的时域 $D_T$ 中工作的，它们只能处理离散序列或是数值流。因此，连续的时域信号必须先转化为离散时域信号，这就是使用**采样保持电路**的目的。图 3.4 左图展示了一个简单的采样保持电路。

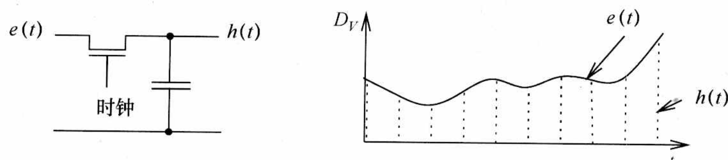



图 3.4　采样保持电路

图 3.4 中的电路包含了一个受时钟驱动的晶体管以及电容，晶体管工作在开关状态。当晶体管受时钟的驱动而置为导通状态时，电容充电，从而使输入电压$h(t)$接近输入电压 $e(t)$。从晶体管转为导通态后，直到晶体管再次关断，电容将一直处于充电状态。电容两端的电压值可以看作来自连续函数 $e(t)$ 的一个离散值序列 $h(t)$（见图3.4 右图）。如果在 $\{t_s\}$ 内的各个时间点上对 $e(t)$ 进行采样，则$h(t)$也仅在这些时间点上有意义。

对于理想化的采样保持电路，它应该可以在任意短暂的时间内改变电容两侧的电压值。只有这样，输出序列中的瞬时电压值才能与电容两端的即时输入电压相匹配。在实践中，电容都有一个较短的时间窗去完成真正的充放电，其电压也会在这个时间窗内有相应的变化。

有趣的问题是：可以从采样得到的信号 $h(t)$ 重构原始信号 $e(t)$ 吗？已经知道，所有的随机信号都可以表示为多个正弦函数的和，这些正弦信号可能有不同相位（可能是相位移动），或者不同频率（傅里叶分解）[⊖]。例如，图3.5 与图3.6 展示了在不同正弦波频率下所近似出的方波，可见在高频率的正弦波时，方波具有更好的平坦度。

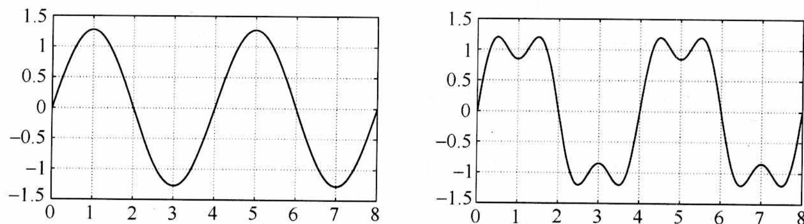



图 3.5　使用 $K=1$（左图）与 $K=3$（右图）的正弦波来近似方波

⊖　本书假定在嵌入式系统课程中没有关于傅里叶分解的完整理论介绍。通过对本书中例子的分析，学生应该可以更好地理解这一理论。

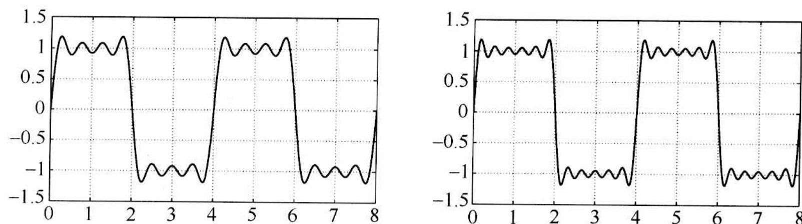


图3.6　使用$K=7$（左图）与$K=11$（右图）的正弦波来近似方波

这些图是对式（3.1）的图形化表示［Oppenheimet al.，2009］，其中$p$是周期：

$$e'_K(t) = \sum_{k=1,3,5,7,9\dots}^{K}\left(\frac{4}{\pi k}\sin\left(\frac{2\pi kt}{p}\right)\right) \tag{3.1}$$

以$Tr$标识的数据处理被认为是线性的（Linear），对于信号$e_1(t)$与$e_2(t)$有

$$Tr(e_1+e_2)=Tr(e_1)+Tr(e_2) \tag{3.2}$$

现在规定，其后的内容都是讲述对线性系统的处理。为了回答前面提出的问题，这里研究对正弦波独立采样的影响。

假定输入信号与函数$e_3$或$e_4$对应：

$$e_3(t)=\sin\left(\frac{2\pi t}{8}\right)+0.5\sin\left(\frac{2\pi t}{4}\right) \tag{3.3}$$

$$e_4(t)=\sin\left(\frac{2\pi t}{8}\right)+0.5\sin\left(\frac{2\pi t}{4}\right)+0.5\sin\left(\frac{2\pi t}{1}\right) \tag{3.4}$$

函数中使用的正弦波的周期分别为$p=8$、4和1［这可以从与式（3.1）的对比中看出］。图3.7所示是对这些函数的图形化表达。

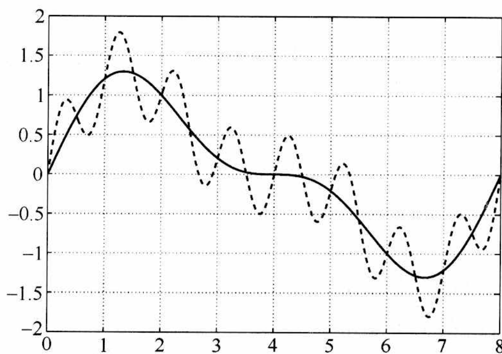


图3.7　函数$e_3(t)$（实线）与$e_4(t)$（虚线）波形

假定都是在整点对这些信号进行采样，则任意点上这两组采样值将相等。很显然，如果只有这一组采样信号，并且按图上的时间点进行采样，则不可能区分$e_3(t)$与$e_4(t)$。通常，对于变化非常缓慢的信号$e_3(t)$或非常迅速的信号$e_4(t)$，如果对两者的采样时间点完全相同，则不可能区分出两者。多个不同信号经采样后，如果有相同的采样值，称为混叠。事实上，对信号$e_4(t)$的采样频率并不足以反应它的变化情况，在两个采样点之间，它还发生了斜率上的变化。从这个反例可知，除非有关于频率或输入信号波形的其他信息，否则对原始信号进行重构基本上是不可能的。

什么样的采样频率才能保证采样不发生混叠呢？

假定以固定的时间间隔来对输入信号进行采样，$p_s$ 是采样周期[㊀]：

$$\forall s: p_s = t_{s+1} - t_s \tag{3.5}$$

令

$$f_s = \frac{1}{p_s} \tag{3.6}$$

为采样速率或采样频率。

根据采样理论［Oppenheim et al. ，2009］，当采样频率大于输入信号频率的两倍时，可以避免混叠：

$$p_s < \frac{p_N}{2} \tag{3.7}$$

$$f_s > 2f_N \tag{3.8}$$

式中　$p_N$——“最快”正弦波的周期；

　　　$f_N$——“最快”正弦波的频率。

**定义**：$f_N$**被称为Nyquist 频率**，$f_s$ 是采样频率。

式（3.8）即**采样定理**，通常也被称为**Nyquist 采样定理**。

因此，只有确保类似 $e_4(t)$ 一样的高频信号从输入中已经被移除，才可能从离散信号 $h(t)$ 中成功重构输入信号 $e(t)$。这也就是抗混叠滤波器的功能。

抗混叠滤波器一般放置在采样保持电路之前（见图3.8）。

图3.9 展示了抗混叠滤波器对输入与输出信号产生的影响。

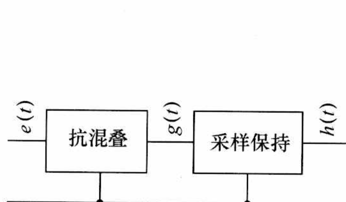



图3.8　在采样保持电路之前的抗混叠滤波器

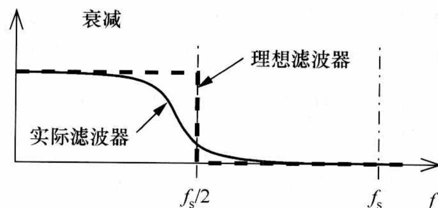



图3.9　理想与实际的抗混叠滤波器（低通滤波器）

理想情况下，抗混叠滤波器应该滤除二分之一采样频率以上的所有频率成分，并且保持其他部分不变。这样信号 $e_4(t)$ 就能转变成信号 $e_3(t)$，但这样的滤波器在现实中并不存在。实际滤波器的频率在小于$f_s/2$ 时即开始衰减，并且它不能消除$f_s/2$ 以上的所有频率成分（见图3.9）。滤波也会对高频成分进行衰减。对于小于$f_s/2$ 的信号，如果某些应用还使用了输入信号的频率参数，可能引起信号“过冲”。抗混叠滤波器的设计，本身就是一门艺术。

㊀ 为了与调度理论中的标识符一致，使用 $p_s$ 来代替 $T_s$。$T_s$ 在数字信号处理中非常常用。

### 3.2.3　数值离散化：A-D 转换器

因为已经约定只讨论数字计算机，因此需要将映射在连续时间内数值域 $D_V$ 中的值，重新映射到离散时间 $D'_V$ 内。将模拟量转变为数字量的工作，即由模-数（A-D）转换器完成的。有许多不同速度/精度的 A-D 转换器，在本书只讨论两种极端情况：

1）Flash A-D 转换器：这种类型的转换器使用了大量的比较器。每个比较器有两个输入，分别记为 + 与 -。如果在 + 端输入的电压超过了 - 端输入的电压，则产生逻辑'1' 输出，否则产生逻辑'0'[⊖]。

在这个 A-D 转换器中，所有的输入都接到一个分压器，如果输入电压 $h(t)$ 超过了$\frac{3}{4}V_{ref}$，则图 3.10a 上面的比较器将产生'1'。比较器输出侧的译码器将把它作为输出值中最高位的'1'。由于 $V_{ref}$一般是电源电压，因此应当尽量避免 $h(t)>V_{ref}$ 的情况，否则可能对电路造成损坏。在这里的例子中，如果转换器没有因为过高的输入电压而损坏，则大于 $V_{ref}$的输入电压将产生最大的数值。

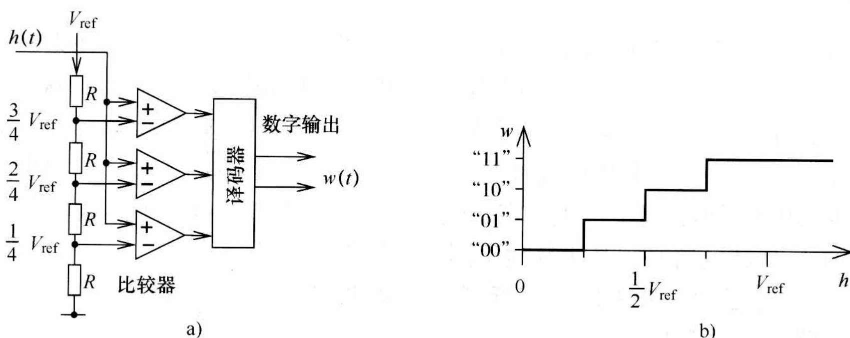



图 3.10　Flash A-D 转换器以及 $w$ 是 $h$ 的函数
a）Flash A-D 转换器　b）$w$ 是 $h$ 的函数

如果输入信号 $h(t)$ 小于$\frac{3}{4}V_{ref}$，但又大于$\frac{2}{4}V_{ref}$，图 3.10 上的比较器将产生输出'0'，而下一个比较器仍然是'1'，编码器将把这个值作为转换结果的次高位。

对于$\frac{1}{4}V_{ref}<h(t)<\frac{2}{4}V_{ref}$和 $0<h(t)<\frac{1}{4}V_{ref}$的情况相似，编码器将产生相应的第三个高位以及最低位。图 3.10b 展示了输入电压与所得数值的关系。

比较器以一种特定的方式来产生输出：如果比较器输出'1'，则它后面的所有

⊖ 在实际中，讨论两个输入完全一致的情况的意义不大。出于多种原因（如温度、制造流程等），这两个输入端的电压总是会有微小的差异。

输出也都等于'1'。编码器将这种数值表达转换为常规的自然数。此处的编码器也常被称为"优先级编码器"，它将最高位为'1'的输入数值转化为二进制数㊀。

这个电路可以将正模拟输入电压转换为数值，如果需要对正负电压同时进行转换并产生互补输出，则需要对电路进行扩展。

A-D转换器的重要指标之一是其分辨率，它有多个不同但仍然相关联的含义[Analog Devices Inc. Eng.，2004]。A-D转换器的分辨率是A-D转换器能够生成的转换结果的位数。例如，16位分辨率的A-D转换器在音频领域有很多应用场景。分辨率也可以以电压来进行量化，此时它表示的是因两端输入之间的差值而使结果增加了1：

$$Q = \frac{V_{FSR}}{n}$$

式中　$V_{FSR}$——最大与最小电压的差值；

$Q$——每一次转换的电压精度；

$n$——电压间隔的数量（不是位数）。

例如，对于图3.10中的A-D转换器，假定$V_{ref}$是最大电压，其分辨率是2bit或$\frac{1}{4}V_{ref}$电压。

Flash A-D转换器的优点在于其转换速度较快，它工作时不会消耗太多的时钟周期。它的输入与输出之间的延时非常小，电路使用起来也比较容易，这样的特性使其很适合在诸如需要高速转换的音频领域应用。Flash A-D转换器的缺点在于其硬件的复杂度，对于$n$个数值需要进行$n-1$次比较才能完全区分。如果用此类A-D转换器在CD唱片机上产生数字音频信号，则需要$2^{16}-1$个比较器！所以，高分辨率的A-D转换器必须基于其他不同的工作方式。

2）**逐次逼近型**：使用逐次逼近型A-D转换器可以实现高分辨率的数—模转换。其电路参考图3.11。

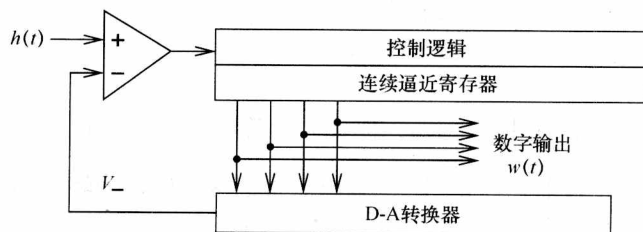



图3.11　逐次逼近型转换电路

逐次逼近型的关键点是使用二进制搜索。在初始阶段，逐次逼近型输出寄存器最高位被设置为'1'，其他位都是'0'。寄存器中的值被转换为一个模拟量，大致

㊀ 这样的编码器在查找浮点数的小数部分为'1'的最高位时，也非常有用。

是0.5×最大输入电压[⊖]。如果 $h(t)$ 大于当前产生的模拟量，则保持寄存器的最高位为'1'，否则将其设置为'0'。

输出寄存器下一个位值总是与前一步的处理相关：如果输入值在输入范围的1/4~1/2之间，则它仍然为'1'。相同的处理一直重复，直到计算出所有的输出位。图3.12展示了这样的例子。

在图3.12中，初始状态时最高位被置为'1'，由于输出的结果 $V_-$ 小于 $h(t)$，则最高位仍然为'1'。而后次高位也被置为'1'，但由于结果 $V_-$ 大于 $h(t)$，这一位被重置为'0'。接下来再对第三个高位进行检查，如此重复。很显然，在整个转换过程中，$h(t)$ 都应该保持不变。对于这样的需求，使用前面所述的采样保持电路即可实现。最后输出的数字信号标识为 $w(t)$。

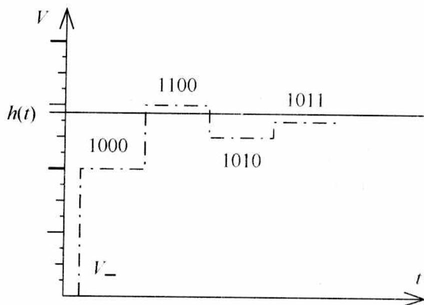



图3.12 逐次逼近型的步骤

逐次逼近型的显著优点是其硬件效率，为了区分 $n$ 个数字量，逐次逼近型转换器的寄存器需要 $\log_2(n)$ 个位，以及 $\log_2(n)$ 个位的D-A转换器。它的缺点在于速度：对于 $n$ 个数字量的转换，它的算法等级为 $O(\log_2(n))$。因此，这种转换器适用于高精度中等速度的场合。

图3.13强调了A-D转换器的行为，在这里，它的输入信号来自式（3.3）。图3.13中仅展示了输入信号为正的部分。

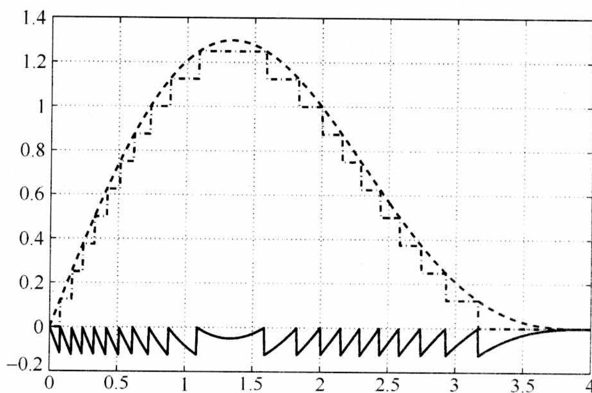



图3.13 $h(t)$（虚线）、输出函数 $w(t)$（点画线）和 $w(t)-h(t)$（实线）

图3.13展示了量化后的数值以及其相应的电压值，同时还包含这两者之间的

⊖ 庆幸的是，从数字量到模拟量的转换（D-A转换）的效率非常高，转换速度也非常快。

差值。很显然，转换器在将模拟信号转换为可用的数值时，采取了“截断”操作（量化数值总是小于或等于模拟值）。“截断”是产生比较结果的一种重要方式。而“凑整”操作则对“半个位”的情况在内部作了校正操作。事实上，数字信号的编码值是原始信号与 $w(t)$ 与 $h(t)$ 差值的和。也就是说，**这两个信号的差值被加入到了原始信号中**，这个差值信号被称为**量化噪声**（Quantization Noise）：

$$\text{量化噪声}(t)=w(t)-h(t) \tag{3.9}$$

$$|\text{量化噪声}(t)|<Q \tag{3.10}$$

很显然，可以通过增加 A-D 转换器的分辨率来降低其量化噪声。量化噪声对转换器的影响可以通过信噪比（Signal-to-Noise Ratio，SNR）的定义来进一步解释。SNR 的度量单位是分贝（1/10 Bel，以 Alexander G. Bell 命名）：

$$\text{SNR}(\text{dB})=10\log\frac{\text{有用信号功率}}{\text{噪声信号功率}} \tag{3.11}$$

$$=20\log\frac{\text{有用信号电压}}{\text{噪声信号电压}} \tag{3.12}$$

在这种情况下，对于给定的阻抗 $R$，信号的功耗是电压的二次方。dB 并不是一个物理单位，因为信噪比并不是量纲。

对于给定的信号 $h(t)$，量化噪声的功耗等于 $\alpha Q$，其中根据 $h(t)$ 可知 $\alpha\leqslant 1$。如果 $h(t)$ 总是可以使用一个数字量准确表达，则 $\alpha=0$。如果 $h(t)$ 总是比下一个值“仅小一点点”，则 $\alpha$ 可能接近 1。

举例（对于 $\alpha\sim 1$），16bit CD 音频的 SNR 可以按下式计算：

$$20\log\ (2^{16})=96\text{dB}$$

而对于高质量的 24bit CD，可以得到大约为 144 dB 的 SNR。$\alpha<1$ 以及 A-D 转换本身的误差都会对这一结果有影响。

也有许多其他类型的 A-D 转换器，它们在转换速率与精度上均不一样［O’Neill，2006］。当前也有许多自动筛选最合适的转换器的技术［Vogels and Gielen，2003］。

## 3.3　处理单元

### 3.3.1　概述

当前所有的嵌入式系统的工作都必须依赖于电能，它们使用的电能总和通常被称为“能耗”。严格来讲，“能耗”这一术语并不正确，因为电能都会转化成为其他形式的能量，典型的就是热能。对于嵌入式系统，持续可用的能量是系统工作的关键因素。这一点其实早有认识：“电力已经被认为是嵌入式系统的最大限制因素之一”［Eggermont，2002］。电力与能耗的重要性，在嵌入式系统的早期就被认识

到了。在近年的通用计算项目中，能耗问题得到了更多关注，也产生了诸如绿色计算（Green Computing Initiative）之类的新概念。对于嵌入式系统中的信息处理，将考虑使用硬连线的可重用 ASIC、可重配逻辑以及可编程序处理器。从能耗的角度考虑，这 3 种技术是非常不同的。图 3.14 重复了图 1.4。

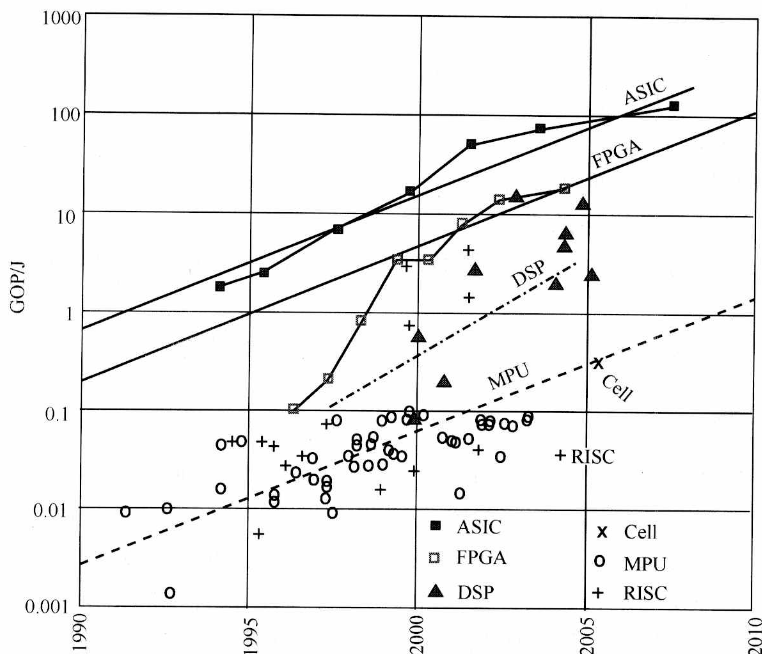



图 3.14　硬件效率（© De Man and Philips）

图 3.14 展示了当前可用的硬件技术中能耗与灵活性之间的矛盾：如果设计目标是低能耗的系统，则应该使用 ASIC 去替代基于处理器或可重配逻辑单元；如果设计目标是灵活性，则显然不能再同时得到低能耗。

对于单次的操作，应用消耗的能量 $E$ 与所需能量 $P$ 之间有下关系：

$$E = \int P\mathrm{d}t \tag{3.13}$$

假定某个系统所消耗的电力为 $P_0(t)$，在经历 $t_0$ 个单位的执行时间后，它所产生的能量消耗为

$$E_0 = \int_0^{t_0} P_0(t)\,\mathrm{d}t$$

假定系统经修改后，它需要 $t_1$ 的时间完成相应操作，其能量消耗为 $P_1(t)$：

$$E_1 = \int_0^{t_1} P_1(t)\,\mathrm{d}t$$

如果 $P_1(t)$ 比 $P_0(t)$ 略大，则减少的执行时间同样会减少能量消耗。但通常这样的结论并不正确：由图 3.15 可见，$E_1$ 可能小于 $E_0$，但 $E_1$ 也有可能大于 $E_0$。

降低电力与能量的消耗都非常重要。电力消耗对供电模块有较大影响，如所选择的电压调整器、布线的复杂度以及散热装置等。尤其是在移动市场领域，对低能耗有着苛刻的要求，这源于电池技术近年来发展缓慢［ITRS Organization，2009］，同时能源的价格也在不断上涨。同时，低能耗的系统也会减少对散热设备的要求，它的稳定性也更好（电子元器件在高温环境下会加速老化）。

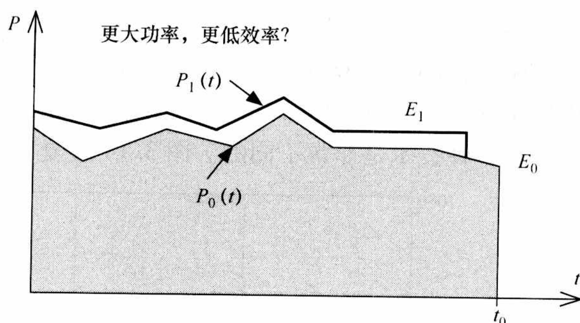



图 3.15　能耗 $E_0$ 与 $E_1$ 的比较

下面首先来考虑一下 ASIC（专用集成电路）中的情况。

## 3.3.2　ASIC

对于高性能应用以及大规模市场，可以考虑在设计中使用 ASIC。但 ASIC 的设计与制造成本都非常高昂：如将光刻模型转成芯片的掩膜工艺，它的一般成本大概是 $10^5$ ~ $10^6$ 欧元或美元，近年来，掩膜工艺的成本更是以指数级上升。但是，这种实现也有着开发周期长、缺少灵活性等缺点：纠正设计中的错误一般需要新的掩膜，并且需要重新加工制造。因此，ASIC 只有适用于那些对能耗有极其苛刻场景，或者市场能接受其成本，又或者系统有着大规模的应用市场。鉴于它的应用范围，本书不对 ASIC 作过多讨论。

## 3.3.3　处理器

处理器的关键优点是其灵活性。使用处理器，就可以通过改变系统软件来改变嵌入式系统的所有行为。改变嵌入式系统的行为可能有多种原因，如修正设计中的错误、更新系统、标准的改变，或者在系统中增加新的功能。正是因为如此，处理器在嵌入式系统中的应用越来越广泛。

嵌入式系统一般要求高效率，它们的指令不需要与通用的个人计算机（PC）兼容。因此，它们的架构可能与 PC 上的处理器有较大差异。能耗有多个不同的衡量因素：

1）能效（Energy-effciency）：系统的架构必须要考虑如何来提高能效，而在软件设计方面也要考虑如何不增加额外的能耗。举个例子，如果因为编译器而使软件产生了 50% 的额外开销，为了仍然能满足系统的运行时间，可能会增加供电电压，或者系统的时钟频率，此时，系统与 ASIC 所能达到的能耗标准差异就更大，可能远不止 50%。

有多种可行的技术可以在各个层面去提高处理器的能效，从软件开发到芯片的

制造流程 [Burd and Brodersen, 2003]，它们都应该被顾及到。门控时钟就是这些提高能效的技术之一。在处理器空闲时，处理器的部分模块将从时钟源断开。例如当 DMA（Direct Memory Access，直接内存存取）或总线桥接器空闲时，它们的时钟将关闭，同时也尽量减少处理器需要的时钟。有两种实现方法：全局同步但局部异步型处理器；全局异步但局部同步型处理器（GALS） [Iyer and Marculescu, 2002]。在 E. Macii [Macii, 2004] 的著作以及 PATMOS 会议论文中（参见 [Monteiro and van Leuken, 2010]），提供了关于低功耗设计的详细技术资料。

有两种技术可以用于较高层的抽象实现：

① **动态电源管理（Dynamic Power Management，DPM）**：基于这种技术，处理器有多个省电状态而不是一直处于标准工作模式。每一种省电状态都有不同的电力消耗，并且转换到正常的工作模式也需要不同的转换时间。图 3.16 展示了 StrongArm SA 1100 处理器的 3 个状态。

在运行状态，处理器全速工作；在空闲状态，处理器仅监听中断输入；在睡眠状态，芯片处于关断状态，处理器被复位，同时芯片的供电也被关断 [Wolf, 2001]。使用独立的 I/O 供电使系统可通过硬件来对电源进行管理，通过预置的唤醒事件，电源管理硬件也可以对处理器进行唤醒操作，使处理器再次进入到运行状态。需要注意的是，睡眠状态与其他状态的电源消耗有很大差异，从睡眠到运行状态存在较大的延迟。

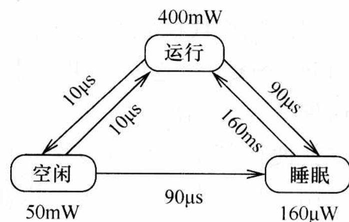


图 3.16　StrongArm SA 1100 处理器的动态电源管理 [Benini et al., 2000]

② **动态电压调整（Dynamic Voltage Scaling，DVS）**：这种方式基于 CMOS 型处理器的能耗将以供电电压 $V_{\mathrm{dd}}$ 的二次方数上升的理论。CMOS 电路的功耗由下式决定 [Chandrakasan et al., 1992]：

$$P = \alpha C_{\mathrm{L}} V_{\mathrm{dd}}^2 f \tag{3.14}$$

式中　$\alpha$——活动的开关数；

$C_{\mathrm{L}}$——负载电容；

$V_{\mathrm{dd}}$——供电电压；

$f$——时钟频率。

CMOS 电路的延迟可以按下式估计 [Chandrakasan et al., 1992]，[Chandrakasan et al., 1995]：

$$\tau = kC_{\mathrm{L}} \frac{V_{\mathrm{dd}}}{(V_{\mathrm{dd}} - V_{\mathrm{t}})^2} \tag{3.15}$$

式中　$k$——常量；

$V_{\mathrm{t}}$——门限电压，$V_{\mathrm{t}}$ 影响着晶体管所需的开启电压。

假定最大供电电压 $V_{dd}$ 为3.3V，$V_t$ 是0.8V。最大的时钟频率是与供电电压相关的函数。减小供电电压将使能耗以二次方数下降，算法的运行时间将以线性的方式增长（不考虑内存系统的影响）。这就可以解释动态电压调整（Dynamic Voltage Scaling，DVS）技术的应用。如Transmeta［Klaiber，2000］的Crusoe™处理器在1.1～1.6V的供电电压之间提供了32个可选挡位，相应的时钟频率也可以按33MHz的步长从200～700MHz变化，从一组电压/频率转到下一组大约需要20ms。在Burd与Brodersen［Burd and Brodersen，2000］的一篇论文中，详细地描述了DVS处理器的设计相关问题。根据这篇论文的论述，未来也许可以通过降低最大 $V_{dd}$，同时降低门限电压（但不幸的是，这可能会增加漏电流，也可能会增加待机时的功耗）来完成。在2004年，Intel® SpeedStep™在Pentium® M处理器上提供了6种可变的速度/电压，从而降低了系统功耗。

2）**代码大小**：由于嵌入式系统一般不外接硬盘驱动器，它的可用内存也比较有限，因此减小代码大小对于嵌入式系统也非常重要㊀。对于片上系统（Systems on a Chip，SoC），这一点更为重要：对于SoC，其内存与处理器是在同一块芯片上，这种情况下的内存一般被称为片内内存。由于片内内存的制造要与处理器兼容，这就使得其工艺更加复杂，成本也非常高昂，片内内存也有可能占去芯片的大部分可用面积。有多种技术可以用于改善代码大小：

① **CISC架构**：标准的RISC处理器更倾向于运行速度，而不是代码大小。而早期的复杂指令集（CISC）处理器一般与低速内存相连，通常也不使用高速缓存，它则更倾向于代码大小的优化。这种“过时”的CISC处理器仍然可以在嵌入式系统中找到，如基于CISC架构的Motorola 68000系列的ColdFire处理器［Freescale semiconductor，2005］。

② **压缩技术**：为了减小指令寻址的开销，从而减少对硅片面积的需求，指令通常都以压缩格式存储在内存中，这种做法同时也降低了存取指令的功耗。使用压缩技术，即使减少带宽，存取指令同样可以做到很快。译码器（希望它是占硅片面积小而且速度小的模块）处于处理器与内存之间，它将压缩指令还原为原始指令（见图3.17右图）㊁。存储经压缩的指令，而不是未压缩的原始指令，这样就减少了对内存的

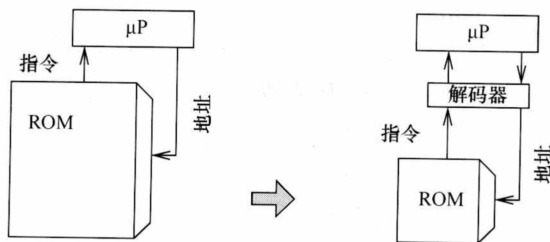



图3.17 压缩指令的解压缩

㊀ 大型闪存类内存的使用，减小了对代码大小方面的约束。

㊁ 依然使用方框来表示多路复用器、算法单元以及内存，它们在本书中都会被广泛使用。对于内存，使用含一个地址译码器的模块（对ROM也适用）来表示，其中的译码器也指出了地址的输入。

需求。

压缩的目的可以总结如下：

a. 节省片内的 ROM 与 RAM，因为它们可能比处理器要昂贵。

b. 可以对指令使用一些译码技术，对于具有以下属性的数据，也可以使用译码：

a）数据对操作延时不敏感；

b）译码应该在有限的空间中工作（例如，它不可能在整个代码中去查找一条分支指令的目的地）；

c）需要考虑到内存、指令与取址的字长；

d）必须支持分支指令的随机寻址；

e）快速编码仅在有可写的数据需要被编码时才需要，否则快速译码就足够了。

减小代码量的技术还有许多变异：

a. 某些处理器还有第二类指令集（Second Instruction Set）。第二类指令集有更精简的指令格式。ARM 处理器系列就是存在第二类指令集的典型例子。ARM 指令集是包含预测指令的 32 位指令集，当且仅当满足某些条件的前提下，相应的指令才会得到执行。指令中提及的条件是指令格式中的最高 4 位。大部分的 ARM 处理器也提供了第二类指令集，它使用 16 位宽，称之为 THUMB 指令集。由于不需要支持预测指令，THUMB 指令集更加精简，它使用更少的操作数以及寄存器（见图 3. 18）。

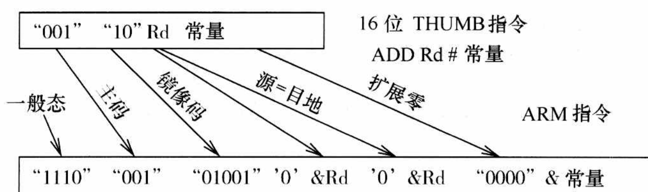



图 3. 18　THUMB 重编码为 ARM 指令

在程序执行时，THUMB 指令可以动态地转到 ARM 指令继续执行。由于它的算法指令只使用一半寄存器，因此 THUMB 指令的寄存器位域总是与一个'0'位相连[㊀]。THUMB 指令的源寄存器与目的寄存器是完全一样的，常量的长度被减小到了 4bit。在指令执行中，使用了流水线来提高运行时的效率。

某些其他的处理器也有类似的技术。这种实现的缺点在于必须对工具（编译器、汇编器、调试器等）进行扩展，这样才能支持第二类指令集。因此，为了支

㊀ 这里使用了 VHDL 的符号，图 3. 18 中用一个 & 符号来表示连接，其后的引号中跟着常量。

持第二类指令集，可能会增加软件开发的成本。

b. 第二种技术是使用字典（Dictionaries）。使用这种技术，每种形式的指令都被惟一存储，对于不同的程序计数器值（即PC），通过一个查找表，在指令空间，也就是在字典中去查找相应的指令（见图3.19）。

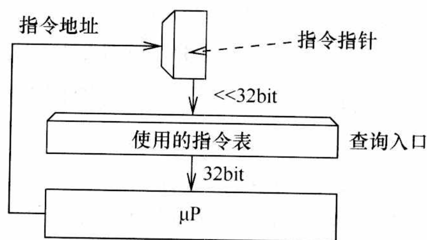



图3.19　指令压缩中的字典方式

这种技术首先是基于只使用了较少的指令数量的前提，只有这样，在指令查找表中才不会有太多的查找入口，指令的位宽也才会比较短。这种技术也有许多变异：如双级控制存储（Two-level Control Store）[Dasgupta，1979]、微程序设计（Nanoprogramming）[Stritter and Gunter，1979]或非队内程序（procedure Ex-lining）[Vahid，1995]。

Beszedes [Beszedes，2003] 与 Latendresse [Latendresse，2004] 在其论文中对已知的压缩技术提供了简要介绍。

3）运行时效：为了在不提高时钟频率的前提下尽量满足软件的运行时间要求，在某些应用领域可以有针对性地选用处理器，如数字信号处理器（DSP），更进一步，可以设计特定应用指令集处理器（Application-Specific Instruction set Processors，ASIP）。作为特定领域的处理器代表，下面考虑一下DSP。在数字信号处理中，数字滤波是常有的操作之一。假定将图3.4㊀所示的滤波处理展开，同时对其中的信号进行转换，如图3.20所示。

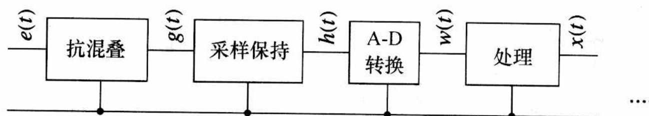



图3.20　信号的转换

式（3.16）描述了从输入信号 $w(t)$ 经过数字滤波，而后产生输出信号 $x(t)$ 的过程。这两个信号都在时间 $\{t_s\}$ 内有定义。为简单起见，用 $x_s$ 代替 $x(t_s)$，用 $w_s$ 代替 $w(t_s)$。

$$x_s = \sum_{k=0}^{n-1} w_{s-k} * a_k \tag{3.16}$$

输出的信号 $x_s$ 是信号 $w$ 的后 $n$ 个信号的加权平均，它可以按每次在当前结果中加上乘积的方式进行迭代计算。对于DSP，每一次的迭代仅使用单一指令就可以完成。来看一个例子：图3.21展示了ADSP 2100 DSP的内部架构。

㊀　此处原书似有误，应为图3.2。——译者注

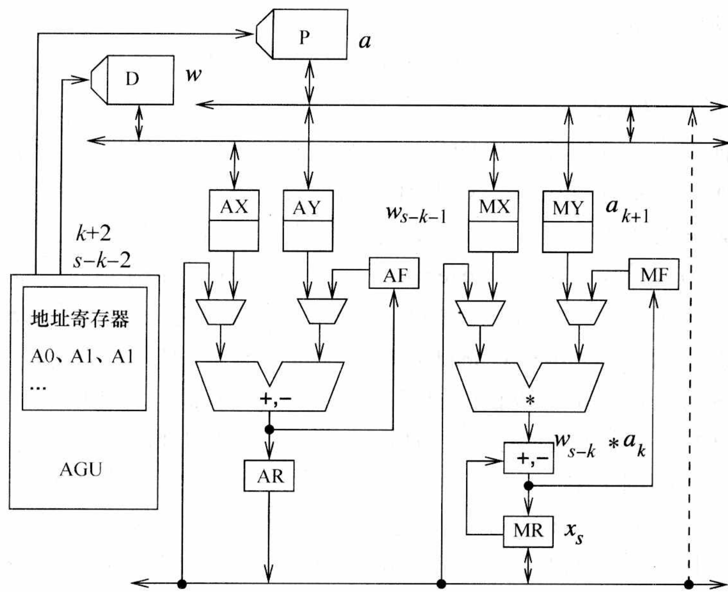



图 3.21　ADSP 2100 DSP 的内部架构

注：此处原书似有误，图中"A0、A1、A1…"应为"A0、A1、A2…"。——译者注

这个处理器有两块内存，称之为 D 与 P。地址产生单元（Address Generating Unit，AGU）用于提供访问内存的指针。加法与乘法有不同的处理单元，它们都有自己的参数寄存器 AX、AY、AF、MX、MY 与 MF。在乘法器的下面又连接了一个加法器，从而更快地对乘加操作进行计算。

对于这个处理器，单次循环一般都只需要一个时钟周期。因此，内存单元 D 与 P 分别被用于存储数组 $w$ 与 $a$，为了使相关的指针能在 AGU 中得到更快更新，也需要占用一些地址寄存器。操作过程中的部分和将存储在 MR 中，计算的流水线使用了寄存器 A1、A2、MX 和 MY。

采样时间 $t_s$ 内的外层循环状况如下：

```
{ MR:=0; A1:=1; A2:=s-1; MX:=w[s]; MY:=a[0];
  for (k=0; k <= (n - 1); k++)
    {MR:=MR + MX * MY;  MX:=w[A2];  MY:=a[A1];
     A1++; A2--; }
  x[s]:=MR;}
```

外层循环与处理时间是对应的。内部的循环体将被编码成单一指令，包含如下的操作：

① 从 MX 与 MY 两个参数寄存器中读取参数，将两个参数相乘而后与部分和相加，再存储到部分和寄存器中；

② 从内存 P 与 D 中读取数据 $a$ 与 $w$ 的下一个元素，而后将它们存储到相应的

参数寄存器 MX 与 MY 中；

③ 更新指向下一个参数的指针，并存储到地址寄存器 A1 与 A2 中；

④ 循环最后的测试。

使用这种方式，内层循环中的每一次迭代都只需要一条指令。为了实现这一点，多个操作也需要并行执行。对于给定的计算需求，这种（有限的）并行化降低了对时钟频率的要求。更进一步的，这种架构中的寄存器完成的是不同的功能，它们可以被称为是**异构的**（Heterogeneous）。对于 DSP，异构的寄存器文件是其很常见的特征。为了避免在循环尾部的测试带来额外的系统开销，在 DSP 上也常常使用**零开销循环指令**（Zero-Overhead Loop Instructions）。使用这种指令，在一个固定的时间段内可以执行单个或较少的多条指令。而没有此类优化的 DSP，为了完成循环内部的迭代，将需要多条指令，从而它有可能需要更高的时钟频率。

在上面描述的例子中，如果 $\{t_s\}$ 是无界的，则数组 $w$ 与 $x$ 也将需要无限大小。由于只有需要 $n$ 个最近的值，因此可以去按需要来约束这些数组的大小。同时，使用模寻址方式，可以对这些数组的空间进行重用（见下面的部分）。

#### 3.3.3.1 DSP

为了在循环体中以一条指令来实现上面滤波功能，DSP 针对不同的应用领域提供了一系列特性：

1）**特定的寻址方式**：在上面描述的滤波器设计中，只需要使用信号 $w$ 的最后 $n$ 个元素，这可以使用环形缓冲区。使用模寻址方式（Modulo Addressing）可以很容易地做到对环形缓冲区的访问。在模寻址方式中，只有访问缓冲区中的第一个或者是最后一个元素，地址才会增加或者减少，除此之外的地址增减，将会导致地址指针指向缓冲区的其他结束位置。

2）**独立的地址产生单元**：地址产生单元（Address Generation Units，AGU）通常与数据存储区的地址输入直接连接（见图 3.22）。

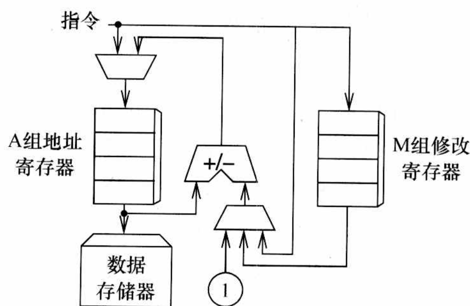



图 3.22 使用特殊地址寄存器的 AGU

在地址寄存器中的地址也可以被用于寄存器间接寻址模式，它可以节省机器指令、运行时间以及能耗。为了提高地址寄存器的利用率，对于用到地址寄存器的大部分指令，都包含着自动递增或自动递减的功能。

3）**饱和算法**：饱和算法改变了计算中的上溢出与下溢出的处理方式。在标准的二进制算法中，在上溢出或下溢出时将使用循环进位方法进行计算。图 3.23 展示了两个 4bit 无符号数相加的情况，可以看出，在标准寄存器中没有返回计算后的进位，而是全零，与真实的计算结果相差也最大。

在饱和算法中，将尽量返回与真实计算结果最接近的数值。在发生上溢出时返回最大值，而在下溢出时返回最小值。这种实现方法对于某些领域尤其重要，如音频或视频应用：用户其实很难区分出真实的计算结果与表达出的最大值引起的差异。另外，因为在实时运行中很难作异常处理，它也可以用在溢出发生的情况中。需要注意的是，在使用饱和算法时，需要清楚正在处理的是有符号还是无符号加法指令，从而返回正确的符号位。

| | |
|---|---|
| | 0111 |
| + | 1001 |
| 标准卷回算法 | 10000 |
| 饱和算法 | 1111 |

图 3.23　无符号整数的循环进位与饱和算法的比较

4）**定点算法**：浮点算法的硬件会增加处理器的成本与功耗。因此据估计，目前大约 80% 的 DSP 均不包含浮点硬件［Aamodt and Chow，2000］。但为了支持整型数，多数处理器都必然支持定点算法。定点数可以从 3 个方面（wl，iwl，sign）来描述，其中 wl 是总字长，iwl 是整数字长（二进制小数点的左侧位数），符号 $s \in \{s, u\}$ 表示了被处理的数是有符号或无符号数。定点数这 3 个方面的关系可参考图 3.24。可以使用的有多种进位模式（如截断）与溢出模式（如饱和算法与循环进位算法）。对于定点数，在乘法操作之后将保持其二进制小数点的位置（一些低位将被截断或循环移位）。对于定点处理器，这一操作是由硬件来支持的。

5）**实时性能**：在现代 PC 中使用的处理器中，都有一些特性去提高程序的平均执行时间。在一些场景中，很难或者根本不可能去验证这些特性是否提高了最坏情况下的执行时间。在某些场景中，甚至最好不要去使用这些特性，如很难保证（也许就是不可能［Absint，2002］）通过使用缓存来对系统进行提速。因此，许多嵌入式处理器都没有缓存，甚至虚拟地址与分页也一般不在嵌入式系统中使用。计算最坏情况执行时间的技术在 5.2.2 节将有更详细的描述。

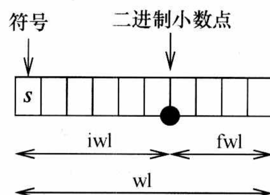



图 3.24　定点数系统的参数

6）**多内存区或多内存**：使用多内存区的好处已经在 ADSP 2100 中得到了证明：两个内存区 D 与 P 允许在同一时刻读取两个参数。多种 DSP 均使用了两个内存区。

7）**异构寄存器文件**：异构寄存器文件已经在滤波器的应用部分提到了。

8）**乘加指令**：这类指令在乘法之后紧随着加法操作，它们也在滤波器应用部分使用到了。

### 3.3.3.2　多媒体处理器/指令集

在许多现代处理器的架构中，寄存器与算法单元至少都是 64 bit 宽的。因此，两个 32 bit 的数据类型（“双字”）、4 个 16 bit 的数据类型（“字”）或 8 个 8 bit 的数据类型（“字节”）可以被一次性存储在单一寄存器中（见图 3.25）。

有较大位宽的算法单元可以在双字、字或字节的边界处更好地处理进位。在此

基础上，多媒体指令也可以对压缩的数据类型进行操作。由于单一指令实现了对多个数据单元的操作，这类指令有时被称为单指令多数据（Single-Instruction Multiple-Data，SIMD）。相比未经压缩的字节存储情况，将多个字节压缩存储到一个64 bit寄存器时最高可将访问速度提高8倍。在内存中的数据一般也以压缩方式进行存储。如果算法指令可以用于压缩的数据类型，则就不需要再对数据进行解压。更进一步地，多媒体指令通常可以与饱和算法结合起来，从而能比标准指令提供更高效的溢出处理方式。因此，使用多媒体指令与压缩存储可以提供比单独使用压缩存储更高效的访问。由于压缩的数据类型带来的好处，在许多处理器中也添加了新的指令。例如，一种被称为**单指令多数据流扩展**（Streaming SIMD Extensions，SSE）的指令已经被添加到了Intel系列的Pentium®兼容处理器中［Intel，2008］。当前（在2010年），Intel® Advanced Vector Extensions（先进向量扩展，AVX）还使用了**短向量指令集**（Short Vector Instructions）［Intel，2010a］。

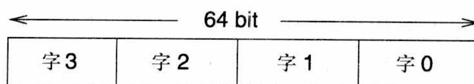


图3.25 存储多个字的64 bit寄存器

#### 3.3.3.3 VLIW处理器

嵌入式系统的计算需求一直在增长，尤其是在包含了高级编码技术或加密的多媒体应用领域。在高性能的微处理器中使用的性能提升技术并不适合于嵌入式系统：它可能需要考虑指令兼容，资源寻找，如在PC中花费大量的资源及功耗去查找应用程序中的并行操作。即使是这样，它们的性能往往仍然不能满足需求。而对于嵌入式系统，首先不必考虑与PC的指令兼容，这样就可以明确使用特定的指令进行并行操作。**并行指令代码**（Explicit Parallelism Instruction set Computers，EPIC）可以完成这一功能。使用EPIC，把对并行操作的查找任务从处理器转移到了编译器，这就在运行过程中节省了芯片内部资源及能量。作为一种特例，下面来考虑甚长指令字（Very Long Instruction Word，VLIW）处理器。对于VLIW处理器，某些操作或指令被共同编码在一个长指令字中（有时被称为**指令包**），这一指令中所有操作的执行是并行的。每一个操作/指令被分别编码成指令包中的一部分，每一部分又控制着相应的硬件单元。如图3.26所示，每一部分都控制着一个硬件单元。

对于VLIW架构，编译器负责产生指令包。这就需要编译器明确可用的硬件单元，从而对它们的使用作出合理的调度。

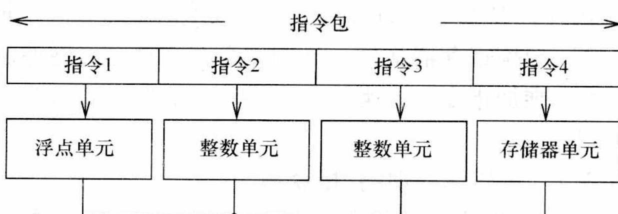


图3.26 VLIW架构（例）

无论功能单元是否在指令周期中被使用，它都必须出现在指令中的相应部分。如果不能对指令包中的各部分进行充分利用，则可能会降低VLIW架构的代码密度。但如果对VLIW添加更多的灵活性，则可以解决这一问

题。如 Texas Instruments TMS 320C6xx 处理器系列的变长指令包，其长度可以达到 256 bit。在指令包中的每一部分，都保留了 1bit 来说明下一部分的指令是否应当并行执行（见图 3.27）。这样，对于未使用的功能单元，就不会有冗余指令的操作。

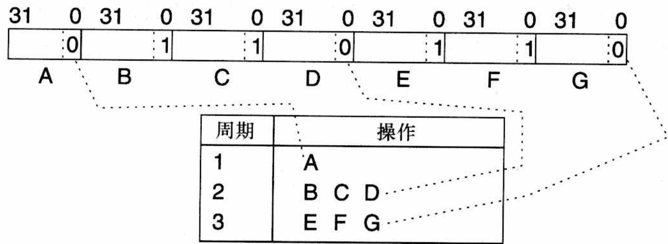



图 3.27　TMS 320C6xx 的指令包

由于使用了变长指令包，TMS 320C6xx 处理器与经典的 VLIW 处理器显得并不一致，但由于它也能明确描述并行操作，因此它仍然是 EPIC 处理器。

1. 分离的寄存器文件

对于 VLIW 与 EPIC 处理器，它们的寄存器文件设计也非常重要。由于并行指令会执行更多的操作，因此在并行指令模式下就需要进行大量的寄存器访问，也就需要大量的访问端口。但是，如果增加处理器内部的端口，则也会增加寄存器的延时、大小以及能耗。因此，许多 VLIW/EPIC 架构都使用分离的寄存器文件。功能单元只与寄存器文件的一部分相连接。如在图 3.28 中展示的 TMS 320C6xx 处理器的内部架构，它包含了两个寄存器文件，每个寄存器文件都与一半的功能单元相连接。对于两个寄存器文件连接到相同功能单元的情况，在一个时钟周期，只允许一个寄存器文件对功能单元进行访问。

Lapinskii 等人［Lapinskii et al.，2001］对寄存器分离方法进行了更多的讨论。

许多 DSP 都是基于 VLIW 架构的，以 M3-DSP 处理器［Fettweis et al.，1998］为例。它包含（相当于）16 个并行数据通路 VLIW 处理器，这些数据通路与一组内存相连，并行地提供处理器需要的参数（见图 3.29）。

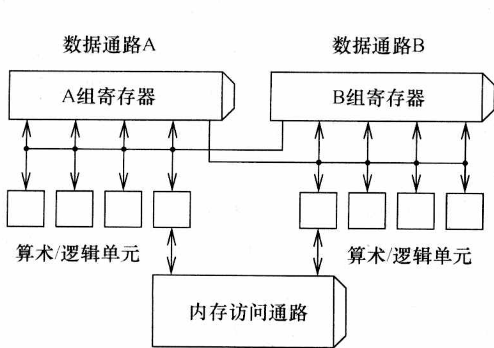



图 3.28　TMS 320C6xx 分离的寄存器文件

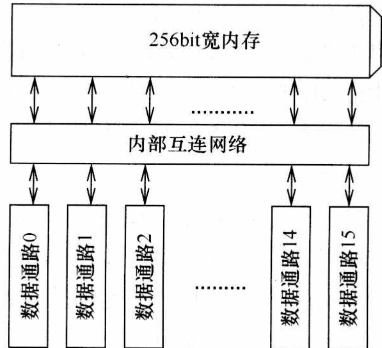



图 3.29　M3-DSP（简图）

2. 预测指令

VLIW 与 EPIC 架构可能有较大的**延时损耗**（Delay Penalty），这是它们的潜在问题之一：这种延时损耗可能产生在一些指令包的跳转指令中。指令包正常时都是按流水线逐一得到执行，流水线的每个阶段仅实现需要执行指令的部分操作，这就意味着在流水线的第一阶段不会发现跳转指令的存在。当跳转指令最后执行完成后，其他指令已经进入了流水线（见图 3.30）。

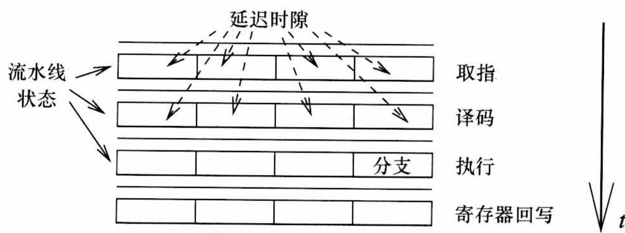



图 3.30　分支指令与延时位置

有两种基本的方法来解决这种情况：

1）将跳转指令按正常指令执行。这种方式被称为**延迟跳转**（Delayed Branch）。在跳转指令之后执行的指令包的位置，被称为**跳转延迟位置**（Branch Delay Slots）。这些跳转延迟位置可以被填充成在跳转指令之前执行的指令。但是，将所有的延迟位置填充成有用的指令是非常困难的事情，有一些延迟位置必须以空操作指令（No-Operation Instructions，NOP）进行填充。术语**跳转延迟损耗**（Branch Delay Penalty）也表达了因为 NOP 的填充导致的性能降低。

2）阻塞流水线直到跳转指令取址完成。在这种情况下不存在跳转延迟位置，它的跳转延迟损耗是因流水线的阻塞而引起的。

有时候跳转延迟会极大地影响系统性能。如 TMS 320C6xx 系列处理器最多可以允许 40 个延迟位置。因此，如果可以避免跳转，则可以提升系统性能。为了避免因 if 判断而引起的跳转，引入了预测指令技术。对于每一条预测指令，都被编码成数个位并在运行时进行检查。如果检查的结果为真，则执行指令，否则转为执行 NOP 指令。在类似于 ARM 的 RISC 处理器中也存在预测指令。如在 3.3.3 节中介绍的 ARM 指令，就包含一个 4bit 的位域，它可以对条件码寄存器中的不同值进行编码。存储在条件码寄存器中的值在运行时都会被检查，它们决定了一系列的指令是否会有影响。

预测指令可以在一些较小的 if 状态下非常有效：if 的条件被存储在其中的一个条件寄存器中，而依赖于 if 条件的主体语句则被作为预测指令执行。这样，if 状态的主体就能与其他操作一起执行而不会引起延迟损耗。

Crusoe（在商业上并没有成功）是为 PC 使用而设计的一种 EPIC 处理器 [Klaiber，2000]。在 PC 上为了应用 EPIC 指令集而做出的努力成就了 Intel 的 IA-64

指令集［Intel，2010b］，它也应用在 Itanium® 处理器上。由于政策上的问题，它的主要应用领域仅在服务器市场。许多片上多处理器系统（MPSoC）（参考 3.3.3.5 节）都是基于 VLIW 与 EPIC 的处理器。

#### 3.3.3.4 微控制器

事实上，嵌入式系统中大量使用的都是微控制器。微控制器往往并不复杂，使用起来也比较容易。由于它们与控制系统设计的相关性，下面介绍一种使用频率最高的处理器，即 Intel 8051，它有如下特性：

1）8 bit CPU，针对控制领域作了大量优化；

2）有大量布尔数据类型的操作指令；

3）64KB 的程序寻址空间；

4）64KB 的独立数据寻址空间；

5）片上有 4KB 的片上程序 RAM，128B 的片上数据 RAM；

6）32 个 I/O，每个 I/O 都可以被独立寻址；

7）片上两个计数器；

8）片上有串行的通用异步接收/传输端口；

9）片上时钟产生器；

10）有许多商业化的变异结构。

对于微控制器，以上这些特性都是比较典型的。

#### 3.3.3.5 MPSoC

提升处理器时钟频率来获取性能提升的方式，近年来已经基本停止使用。处理器使用数 $GH_z$ 的时钟频率带来的高能耗是主要因素之一。为了进一步提升整体性能，有必要使用多个处理器。这使得在芯片设计中，像添加外设器件与内存等附加模块一样，也可以包含多个处理器。以这种方式实现的系统，称之为（MPSoC）。对于通用目的的计算与 PC，多处理器系统通常都是同构（Homogeneous）的（所有处理器都是同种类型）。术语多核（Multi-core）系统都通常与此类系统相关。对于嵌入式系统，能耗需要在设计中被重点考虑。低能耗通常需要通过高度定制的处理器来实现。如在移动通信或图像处理领域，就有许多定制的处理器。图 3.31 包含了 SH-MobileG1 芯片的 Floor-plan［Hattori，2007］。

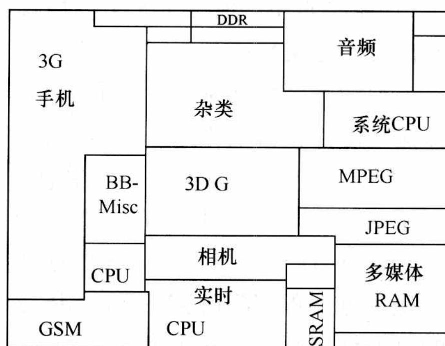



图 3.31　SH-MobileG1 芯片的 Floor-plan

这个芯片展示了高度定制的处理器的使用：对于 MPEG 与 JPEG 编码，GSM 与 3G 移动通信等领域都有特定的处理器。为了节省能量，未使用的区域通常都处于

关断状态。使用顺序语言在多核架构的系统上进行编程是一个挑战，这在第6章将进行讲述。在这类处理器上的应用实现技术非常重要，因为例子展示了它同样可以达到与ASIC接近的能耗情况。举例来说，IMEC的ADRES处理器，它每瓦可以进行大约$55\times10^9$次操作（能耗大约是ASIC的50%）［Man，2007］，［IMEC，2010］。

### 3.3.4　可编程序逻辑

在许多情况下，高度定制的芯片（ASIC）成本过于昂贵，而软件解决方案的效率低且能耗较高。如果算法在定制的硬件上可以高效运行，则也可以考虑使用可编程序逻辑来解决这一问题。可编程序逻辑与ASIC的运行效率几乎一样，它与ASIC不同的是其功能可以被重新编程。基于这些属性，可编程序逻辑可以应用到如下领域：

1）快速原型：当前流行的ASIC都比较复杂，它的设计工作耗费的时间与成本都非常大。因此通常需要先设计一个产品原型，从而来估计它与最终系统的“近似”行为。这一快速原型的成本可以比最终系统要高，体积也可以比最终系统要大，功耗也可以大于最终系统，甚至某些时序约束也可以放宽，它只要能对系统的基本功能进行评估即可。这样的系统也可以用于对系统的最终功能进行评估。

2）小批量应用：如果目标市场的预期销量较小，则无法判断是否应该使用定制的ASIC。如果软件开发较慢或者效率不高，可编程序逻辑就是这种应用场景的明智选择。

3）实时系统：基于FPGA的设计时序通常是非常精确的，因此FPGA可以用于实现时序要求严格的系统。

可编程序硬件通常包含随机存取存储器（Random Access Memory，RAM），用于存储在硬件正常操作时所需要的配置。这些RAM通常都是易失性的（Volatile）（即只有在系统上电后才会存取信息）。因此，在系统上电时，必须将配置数据复制到配置RAM。类似只读存储器（Read-Only Memories，ROM）与闪存的永久（Persistent）存储技术也可以提供这些配置数据的存储功能。

**现场可编程序门阵列**（Field Programmable Gate Arrays，FPGA）是使用最为广泛的可编程序硬件。正如其名字所示，FPGA的编程是“在现场”（在芯片被制造后）进行的。此外，这些可编程器件都包含着阵列化的处理单元。图3.32展示了Xilinx Virtex-II的阵列结构［Xilinx，2007］。

最新的Virtex-5阵列包含了240×108个可配置逻辑模块（Configurable Logic Blocks，CLB）［Xilinx，2009］。它们可以使用可编程序内部连接结构互连。这一阵列可以包含多达1200个用户可定义的I/O连接。此外，它还包含着1056个DSP模块，DSP模块包含25×18bit的乘法器以及16416 kbit的RAM（块RAM）。每个CLB包含着两个slice（见图3.33）。

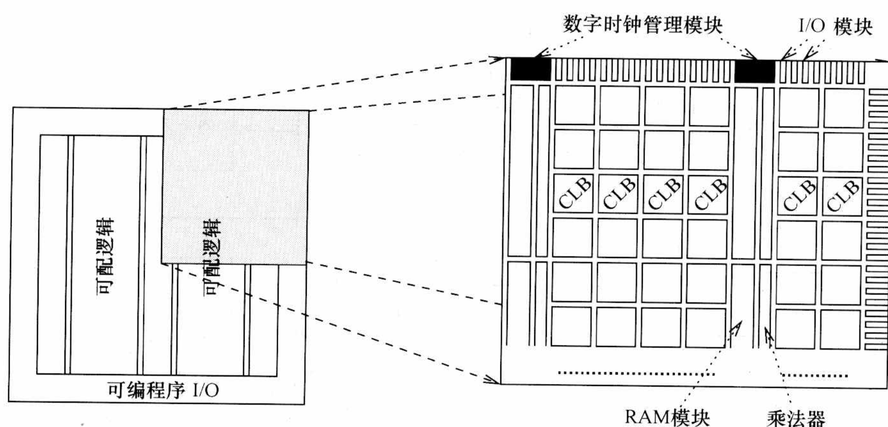



图 3.32 Virtex-II FPGA 的 Floor-plan

每个 slice 包含了 4 块内存。每块内存均可以被当作查找表（Look-Up Table，LUT）使用，它可以实现一个 6 输入的逻辑功能或者两个 5 输入的逻辑功能，对应地可以实现 6 输入的 $2^{64}$ 与 5 输入的 $2^{32}$ 的布尔计算。在多路复用器的基础上，多个内存块还可以组合在一起使用。内存块也可以用作普通的 RAM 或者是移位寄存器（Shift Registers，SRL[⊖]）。每个 slice 还包含 4 个输出寄存器以及针对快速加法的特殊逻辑（见图 3.34）[Xilinx，2009]。

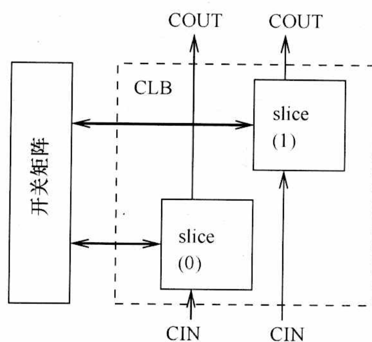



图 3.33 Virtex-5 的 CLB

配置数据决定了 slice 中多路复用器的设置、寄存器与 RAM 的时钟、RAM 模块的内容以及多个 CLB 之间的连接关系。通常，这些配置数据是从硬件功能的高层描述中产生的，如 VHDL。在较为理想的情况下，相同的硬件描述也可以被用于自动产生 ASIC，但在实践中，一般都需要人工干预这一工程。

如果可以在 FPGA 中使用处理器，则将处理器与可配置软件进行集成，就会是一件非常简单的事情，它们可以做成硬核或软核。对于硬核，将在架构中的特殊区域包含一个高度集成的核，这一区域只能用于硬核；而软核可以利用标准的 CLB 来进行综合实现。软核虽然更加灵活，但它的效率比硬核低。

如 Xilinx 的 Virtex-5FXT 系列就包含了最多两个 Power-PC 处理器硬核。

软核可以直接在 FPGA 片上实现，MicroBlaze［Xilinx，2008］就是这样一个例子。

⊖ 此处原书有误“SRL”应改为“SR”。——译者注

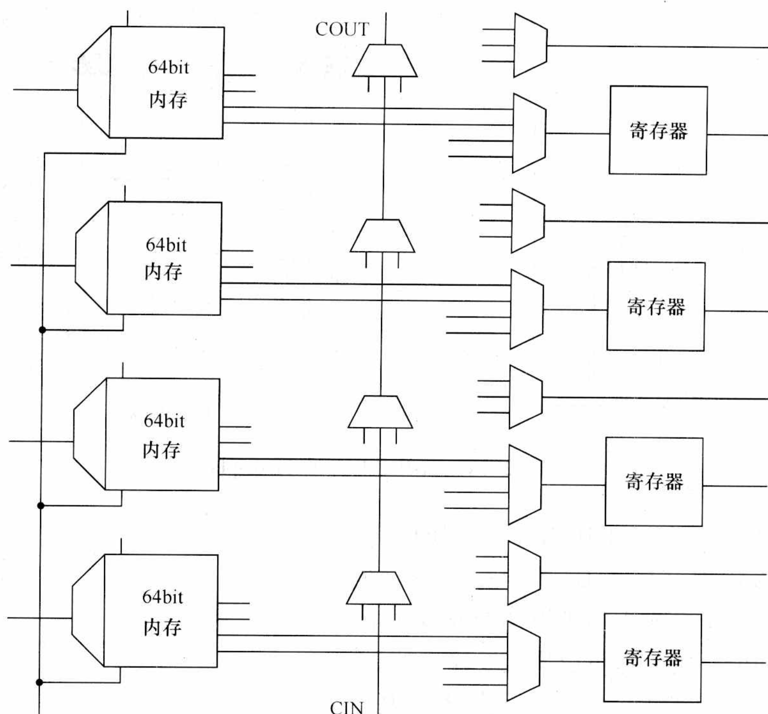



图 3.34 Virtex-5 的 slice（简化图）

## 3.4 内存

数据、程序以及 FPGA 的配置数据都需要被存储在某些内存中，它们必须以一种高效的方式存储。高效，则意味着实时性、代码大小以及能耗方面都更优化。代码大小需要一个好的编码器支持，同时也可能需要代码压缩技术的支持。设计时需要明确内存的架构，从而得到更佳的实时性以及能效，其根本原因是越大的内存就需要越高的功耗，同时它的访问速度也比小内存慢。

图 3.35 分别展示了在将一块内存作为寄存器存储区域时，寄存器访问所需要的时间与功耗以及内存大小的函数关系 [Rixner et al.，2000]。

使用高速缓存的功耗与时延可以使用 CACTI [Wilton and Jouppi，1996] 进行计算，在其计算结果中，也会包含数据 RAM 区域的功耗与时延。这些数值可以在系统设计中用于对通用 RAM 型内存的功耗与时延进行估计。图 3.36 展示了在不同内存大小的情况下，功耗与时延的函数关系图 [Banakar et al.，2002]。

当前的研究已经表明，CPU 与内存的性能在近年来以不同的速度在增长（见图 3.37）。

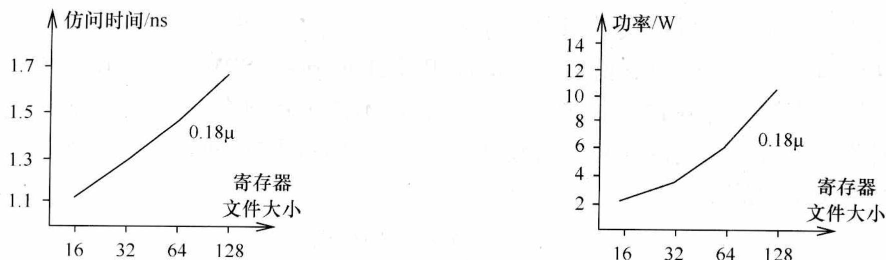



图 3. 35　寄存器文件大小与访问时间及功耗的函数关系

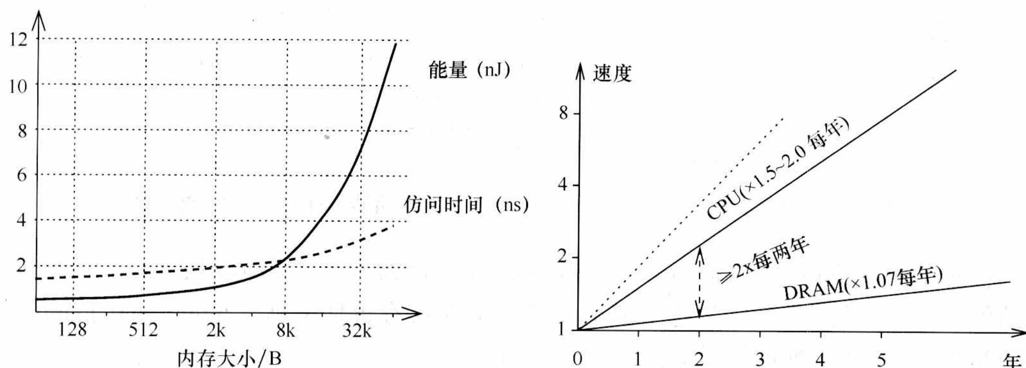



图 3. 36　CACTI 估计的 RAM 内存的功耗及时延

图 3. 37　处理器与内存发展的差距

从图 3. 37 可见，内存的性能每年仅增长 1. 07 倍，而处理器的性能却以每年 1. 5 ~2 倍的速度增长［Machanik，2002］。这意味着处理器与内存之间的性能差距越来越大。当然，更进一步的提升处理器性能可能需要使用多核系统（Multi-Core Systems）。

因此，在主内存与处理器之间使用较小的高速内存作为缓存就非常重要。与 PC 一类的系统一样，这些空间较小的内存必须从架构上来保证系统的实时性是可预测的。在系统中通常使用小内存与大内存的组合，其中的小内存保存着被经常使用数据与指令，大内存保存着其他数据与指令，这样的组合也比使用单一的大内存效率更高。如 A. Macii ［Macii et al. ，2002］所说，在有些时候，也可以考虑使用内存分区。

为了提供更好的实时效率，也常常在系统中使用缓存。在图 3. 35 右图中，可以明显地看出使用缓存时提高了内存系统的能效。对缓存的访问，其实就是对小内存的访问，因此相比大内存而言，它需要更少的功耗。但是，对于被访问的地址，硬件需要去检查缓存中是否包含了被访问地址以及地址中的内容是否有效。这种检查是通过比较缓存的标识（Tag）区域中的相关地址位来完成的［Hennessy and Patterson，2002］。读取这些相关的标识位都需要额外的功耗，因此缓存实时性的可预测性通常也比较低。

另外，小内存可以被映射到大内存的一块空间中（见图3.38）。

这种内存被称为**片上存储器**（Scratch Pad Memories，SPM）。使用频率较高的变量与指令都被分配在这一地址空间中，它也不需要由硬件来对访问地址进行检查。因此，它每次访问的功耗相对较低。图3.39展示了在SPM上的单次访问与缓存中的单次访问的功耗对比。

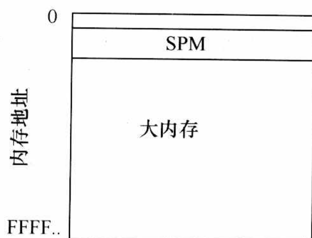



图3.38　SPM示意图

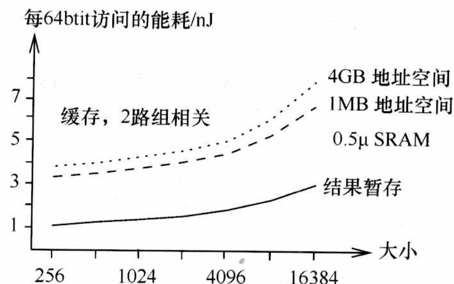



图3.39　片上存储器与缓存访问的能耗对比

对于两路组相关（Two-way Set Associative）的缓存，它与使用SPM时的能耗差值为3。例子中计算出的这个值是使用CACTI缓存评估工具［Wilton and Jouppi，1996］对RAM阵列进行了能耗估计所得到的。

如果编译器能够很好地将经常使用的变量放置在SPM区域，则SPM就可以使内存访问具有更高的可预测性。

## 3.5　通信

在嵌入式系统中，要实现对信息的处理，则首先要准备好信息内容。信息通过不同的**信道**（Channels）进行传输与通信。与通信系统的最大信息传输能力以及噪声等参数一样，信道是通信系统属性的一种抽象。使用通信理论技术，也可以计算出可能的通信错误。通信中使用的物理介质被称为**介质**（Media）。重要的通信介质有：无线介质（射频、红外）、光学介质（光纤）、导线等。

各种不同类型的嵌入式系统也需要多种多样的通信方式。通常，连接不同的嵌入式硬件模块是非常重要的问题，可以对一些通用的需求展开讨论。

### 3.5.1　需求

下面列举了一些必须满足的需求：

1）实时性：这一需求对通信系统的设计有着深远的影响。例如，像以太网（Ethernet）一类低成本的解决方案就不能满足这一需求。

2）效率：连接不同的硬件模块的成本可能会非常高昂。举例来说，大型建筑之间点到点的连接几乎是不可能的。当前已经发现，在汽车的控制单元与外部设备

之间使用单独的导线将会明显增加汽车的成本以及整车重量。使用单独的导线，也将导致很难再添加新的元件。提供低成本设计的需求也同样影响着电力向外部设备供给的方式。通常，在汽车中都会使用一个集成化的供电模块来降低成本。

3）**合理的带宽以及通信延时**：嵌入式系统对带宽的要求各不一样：一方面要提供足够的通信带宽；另一方面也不会导致通信系统成本过高，这是非常重要的。

4）**支持事件驱动型通信**：轮循系统提供了可预测的实时系统行为，但是它们的通信延时可能会非常大。出于对快速基于事件通信的需求，如一些紧急信息可能需要立即通信，它们不可能等待核心控制器来读取这些信息。

5）**健壮性**：信息—物理系统可能被用于某些极端的温度，也可能靠近大量的电磁辐射源等。如汽车引擎，它可能工作在 –20 ~ 180℃（–4 ~ 356℉）的温度下。剧烈的温度变化会影响电压的线性度以及时钟频率的稳定性，同时也会影响通信的稳定性。

6）**容错**：即使在系统的健壮性设计方面付出全部努力，系统仍然可能会产生错误。在系统发生错误后，信息—物理系统应尽可能继续工作。在 PC 中使用的重启，在信息—物理系统中，不可能在故障发生时采用像 PC 一样立即重启，这是不可接受的。这意味着在通信失败时就需要进行重传。这就与第一项需求产生了矛盾：如果允许重传，则满足实时需求可能会比较困难。

7）**可维护性与可诊断性**：非常明显，在有限的时间内修复嵌入式系统也非常重要。

8）**隐私保护**：保密信息可能需要使用加密方式进行传输。

这些通信系统的需求，其实也是在第 1 章中提到的嵌入式/信息—物理系统一般属性的直接描述。这些需求之间可能存在着冲突，因此需要使用一些折中方案。如这些不同的通信模式：一个具有高速带宽的实时系统，同时也没有容错机制（这种模式适合用于流媒体）的系统；一个不可以丢弃短消息的有容错机制的低带宽系统。

## 3.5.2　电气健壮性

对于电气健壮性的设计有许多种技术。芯片内部的数字通信一般都使用一种被称为单端（Single-ended）的信号。对于单端信号，信号在单一的导线上进行传输（见图 3.40）。

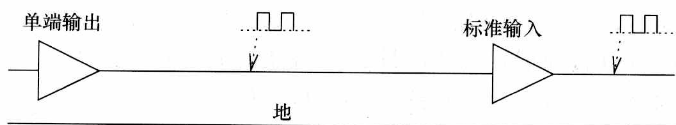



图 3.40　单端信号

这样的信号都使用一个以公共地为参考的电压信号来表达（有少数时候也使

用电流）。对于多个单端信号，使用单一地线即可。单端信号容易受到外部噪声的污染。如果外部噪声（如来自电动机的开关噪声）影响了信号电压，则可能会破坏传输的消息。同时，由于地线之间的阻抗（以及感抗），在有多个节点的通信系统中很难建立一个高质量的共模地。这与差分信号的情况又不一样：对于差分信号，每个信号都需要两条线（见图3.41）。

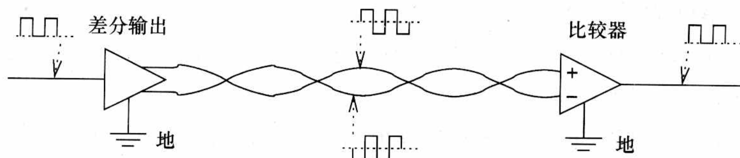



图3.41 差分信号

使用差分信号，二进制数值按如下规则进行编码：如果第一根线上的电压相对于第二根线是正极性的，则它的编码为'1'，否则它被编码成'0'。这两条线通常绞在一起，称为双绞线（Twisted Pairs）。这两条线都可以共同使用局部地信号，非零的局部地信号并不会对系统带来负面影响。差分信号的优点还包括：

1）噪声将以同样的方式影响差分信号的两条传输线，但比较器会去除几乎所有的噪声。

2）比较器的逻辑输出仅与两根线之间的相对电压极性相关。电压的幅度可能会受信号反射以及两条线之间阻抗的影响，但它不会影响到最后的编码结果。

3）由于信号不会在地线上产生任何电流，因此地线的质量不是非常重要。

4）由于不需要在多个通信节点之间建立一个高质量的参考地，因此不需要共模地线（这也就是Ethernet的传输使用差分信号的原因）。

5）由上面列举的诸多关于差分信号的属性，使用差分信号时的数据吞吐量可以比单端信号更大。

但是，差分信号都需要两条线，并且它同时还需要负电压（除非它基于有补充逻辑信息的单端信号电压）。

差分信号已经被广泛使用，如在标准的基于Ethernet的网络中。

### 3.5.3 实时性的保证

对于内部通信，系统可以使用点到点（Point-to-Point）通信或者共享总线通信。点到点通信可以做到很好的实时性，但可能会需要较多的连接，同时在接收端可能会有较多的冲突。对于共享总线，通常写操作更容易。典型地，如果多个发起者都请求访问同一介质（参考如［Hennessy and Patterson，2002］），这些总线一般使用基于优先级的仲裁方式。由于在设计阶段很难去估计系统中的冲突，这种基于优先级的仲裁方式就比较缺乏时序的可预测性。它有时还会导致“饥饿”（低优先级的通信一直被高优先级的通信阻塞）。为了解决这类问题，可以使用时分多址

（Time Division Multiple Access，TDMA）技术。在 TDMA 调度中，每个节点都被分配一个固定的时隙（Time Slot）。通信时间通常被分为帧，每个帧都有一些时隙用于帧同步，同时它也允许发送端在某些时隙关闭通信（见图 3.42 [Koopman and Upender，1995]）。

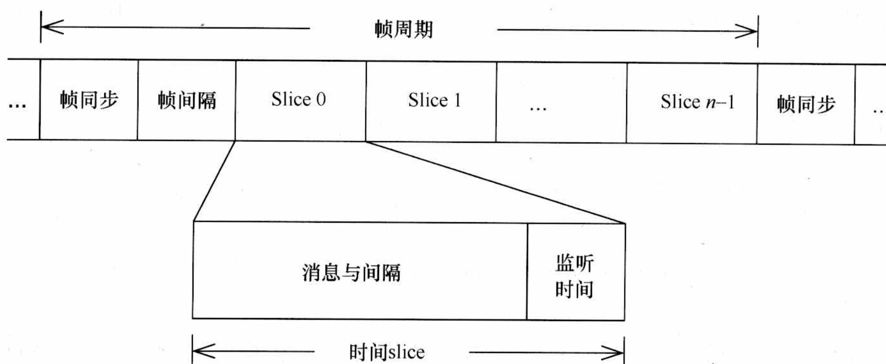



图 3.42　基于 TDMA 的通信

消息间隙的后面还有许多的时间片，它们都可以用于消息通信。考虑到参与通信的各个节点可能有不同的时钟频率，每个时间片也包含一些间隙以及监听时间，时间片被分配给不同的通信节点。这种调度方式有许多变异，如可以将某些未使用的时间片去掉，或者在保证所有节点带宽的前提下为一个节点分配多个时间片。这种调度方式可以避免饥饿。ARM 使用的 AMBA-bus [ARM Ltd.，2009a] 就包含着基于 TDMA 的总线分配模型。

计算机之间的通信通常都是基于 Ethernet 标准的。对于 10 Mbit/s 与 100 Mbit/s 版本的 Ethernet，多个通信节点之间可能会存在冲突。也就是说，多个节点可能在同一时刻发起通信，导线上的信号可能会恶化。只要这种情况一发生，所有的发送端都必须马上停止通信，等待一段时间再进行重传。这个等待时间是随机选择的，从而使下一次的传输尝试不至于再产生新的冲突。这种方式被称为**载波监听多路访问/冲突检测**（Carrier-Sense Multiple Access/Collision Detect，CSMA/CD）。对于 CSMA/CD，可能会存在很小的机率使通信时的冲突在很长一个时间一直存在，从而导致通信延时变得非常大。因此，CSMA/CD 不能用于需要满足实时性约束的系统。

这个问题可以使用**载波监听多路访问/冲突避免**（Carrier-Sense Multiple Access/Collision Avoidance，CSMA/CA）。正如其名称所示，冲突可以被完全避免，而不仅仅是侦测。对于 CSMA/CA，所有节点都被分配了优先级。在**仲裁阶段**（Arbitration Phases），根据通信的不同阶段，通信介质被分配给各个通信节点。在仲裁阶段，准备传输数据的通信节点都需要在介质上出示其优先级标识，而后移除高优先级节点的优先级标识。

为了保证这一点，在仲裁阶段必须有一个时间上限，CSMA/CA将保证具有高优先级的节点能进行可预测的实时传输。对于其他节点，如果更高优先级的节点对通信介质不是连续访问的，则其实时性也可以得到保证。

值得注意的是，高速版本的Ethernet（≥1 Gbit/s）同样也可以避免冲突。TDMA调度同样可以用于无线通信系统中，举例如手机中也使用一些TDMA机制来进行通信，如GSM。

### 3.5.4 例子

1. 传感器/执行器总线

传感器/执行器总线在诸如开关、灯具以及一些处理设备之间提供了通信机制。有许多这样的设备，它们之间的总线的布线成本必须要额外加以考虑。

2. 现场总线

现场总线与传感器/执行器总线相似。通常，它们比传感器/执行器总线能够提供更快的数据速率。现场总线的例子如下：

1）**控制器局域网**（Controller Area Network，CAN）：为了连接控制器及其外设，Bosch与Intel公司在1981年开发了CAN总线。由于CAN可以将许多电缆连接的通信系统用单一总线替代，因此它在汽车工业领域得到了非常广泛的应用。由于汽车市场的规模较大，因此CAN模块都相对比较便宜，它也被用于其他一些领域，如智能家居以及制造设备。CAN总线有如下一些特性：

① 使用双绞线差分传输；

② 基于CSMA/CA的仲裁；

③ 吞吐率为10kbit/s～1 Mbit/s；

④ 消息分为高优先级与低优先级；

⑤ 对于高优先级消息最大时延为134μs；

⑥ 消息编码与PC上的串口（RS-232）相似，都针对差分信号进行了修改。

基于CSMA/CA的仲裁不能防止饥饿，CAN协议也继承了这一缺点。

2）**时间触发协议**（Time-Triggered-Protocol，TTP）[Kopetz and Grunsteidl，1994]用于要求高安全性的容错系统中，如汽车中的安全气囊。

3）**FlexRay** ™[FlexRay Consortium，2002]是一种TDMA协议，它由FlexRay协会（BMW、DaimlerChrysler、General Motors、Ford、Bosch、Motorola以及Philips Semiconductors）开发。FlexRay是TTP以及Byteflight [Byteflight Consortium，2003]的组合变异协议。

FlexRay有静态以及动态的仲裁阶段。静态仲裁阶段使用类似TDMA的仲裁调度，它可以用于实时通信并且可以避免饥饿，动态仲裁可以为非实时通信提供较好的带宽保证。出于容错的考虑，通信节点可以被连接到至多两个总线。**总线监护**（Bus Guardians）可以保护节点免受其他节点大量冗余消息的影响，这种节点发送

大量无用信息的现象被称为babbling idiots。节点可以使用它们自己独立的时钟周期，但所有节点的时钟周期应该互为位数关系。分配给节点用于通信的时隙也正是基于这些具有一定共性的时钟周期。

Levi 仿真软件允许在实验室条件下对此协议进行仿真［Sirocic and Marwedel, 2007a］。

4）LIN（Local Interconnect Network，本地互连网络）是在汽车领域连接传感器与执行器的一种低成本通信标准［LIN Administration, 2010］。

5）MAP：MAP 是专为汽车工厂设计的一种总线。

6）EIB：EIB（European Installation Bus，欧洲安装总线）是为智能家居设计的一种总线。

3. 有线多媒体通信（Wired Multimedia Communication）

对于有线多媒体通信，一般都需要较高的数据传输速率。如MOST（Media Oriented Systems Transport，面向媒体的系统传输）就是汽车领域中用于多媒体及娱乐设备的通信标准［MOST Cooperation, 2010］。如相线接口 IEEE 1394 也可以用于相同的目的。

4. 无线通信（Wireless Communication）

无线通信方式正变得越来越流行。当前（2010 年）使用 HSPA（High Speed Packet Access，高速分组接入）可以很容易得到 7 Mbit/s 的速率，甚至更高速的通信也即将达到［如长期演进（LongTerm Evolution，LTE）技术］。

蓝牙是连接诸如智能手机与手持设备之间的一种通信标准。

无线 Ethernet 的 IEEE 标准版本是 802.11，它已经用在了本地局域网中（Local Area Networks，LAN）。

DECT 是欧洲无线手机的标准。

## 3.6 输出

嵌入式/信息—物理系统的输出设备通常包括：

1）**显示设备**：显示技术也是一个非常重要的领域。相应地，关于显示技术也有大量的相关信息［Society for Display Technology, 2003］。在显示技术的最新研究与发展上，当前最主要的是有机显示技术［Gelsen, 2003］。有机显示是主动发光的，它可以制造得非常轻薄。与 LCD 显示不同，它不再需要背光与偏振滤波器。当前有机显示技术的市场有很广阔的发展前景。

2）**机电设备**：通过电动机或其他一些机电设备，可以对运行环境产生影响。

在实际系统中通常会同时使用到模拟与数字输出。在模拟输出设备中，数字信息需要首先通过 D-A 转换器转换为模拟量。可以在嵌入式系统的模拟输入至输出路径中，找到这一转换器。图 3.43 展示了信号在整个流程中的变化，各个方框表

示的功能与目的将会在接下来的内容进行介绍。

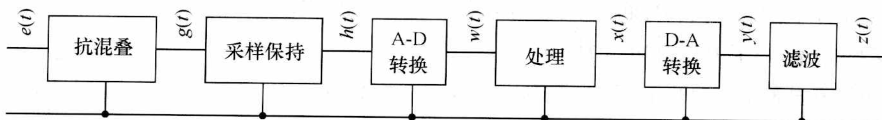



图 3.43　信号从模拟输入到输出的系列转换

## 3.6.1　D-A 转换器

D-A 转换器并不复杂。图 3.44 展示了一个被称为权电阻型的简单 D-A 转换电路。

转换器的关键点是首先根据数字信号 $x$ 的值产生相应的电流信号。但由于电流信号在后续系统中处理时并不方便，因此电流又被转换成相应的电压 $y$。这种转换是通过一个运算放大器（即图 3.44 中的三角形符号）来完成的。关于运算放大器的特征等，在本书的附录 B 中进行了描述。

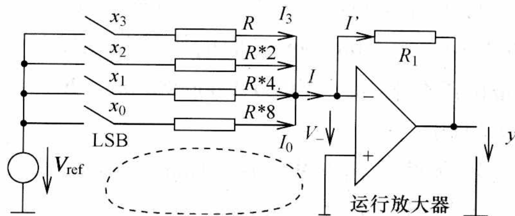



图 3.44　D-A 转换电路

那么，如何来计算输出电压 $y$ 呢？首先，需要考虑图 3.44 虚线表示的回路的意义。数字信号 $x$ 为'0'，则通过电阻的电流也将为零；如果数字信号 $x$ 为'1'，则电流将由每个数字信号的位表示的权重以及电路中选取的电阻来共同决定。对此回路使用基尔霍夫回路定理（Kirchhoff's Loop Rule）（参见附录 B），使用 $x_0$ 来表示 $x$ 的最低位，于是有

$$x_0 \cdot I_0 \cdot 8 \cdot R + V_- - V_{ref} = 0 \tag{3.17}$$

$V_-$ 大约为 0（参见附录 B），因此有

$$I_0 = x_0 * \frac{V_{ref}}{8 * R} \tag{3.18}$$

公式中的 $I_1 \sim I_3$ 是通过其相应电阻的电流值。现在可以基于基尔霍夫回路定理（参见附录 B）对连接了所有电阻的电路节点进行分析。在某个电路节点，所有输出的电流值必须与输入节点的电流值相等，因此有

$$I = I_3 + I_2 + I_1 + I_0 \tag{3.19}$$

$$I = x_3 * \frac{V_{ref}}{R} + x_2 * \frac{V_{ref}}{2 * R} + x_1 * \frac{V_{ref}}{4 * R} + x_0 * \frac{V_{ref}}{8 * R}$$
$$= \frac{V_{ref}}{R} * \sum_{i=0}^{3} x_i * 2^{i-3} \tag{3.20}$$

现在可以对包含 $R_1$、$y$ 与 $V_-$ 的回路应用基尔霍夫回路定理。由于 $V_-$ 约等于0，

因此有

$$y + R_1 * I' = 0 \tag{3.21}$$

接下来，可以对连接着 $I$、$I'$以及运算放大器的反向输入端应用基尔霍夫回路定理。放大器反向输入端的电流可以认为是零，于是电流 $I = I'$，因此有

$$y + R_1 * I = 0 \tag{3.22}$$

从式（3.20）与式（3.22），可以得到

$$y = -V_{\text{ref}} * \frac{R_1}{R} * \sum_{i=0}^{3} x_i * 2^{i-3} = -V_{\text{ref}} * \frac{R_1}{8 * R} * \text{nat}(x) \tag{3.23}$$

式中　nat——数字信号 $x$ 的自然数。

很显然，$y$ 是相应于 $x$ 的数值。如果需要将输出电压全部转为正向，或者要将输出结果进行位翻转，只需要对电路进行很小的扩展即可。

从 DSP 的观点来看，$y(t)$ 是一个离散时域上的信号：它提供了一个在离散时间上的电压值序列（Sequence）。在工程实践中，由于在更多时候都是观察图 3.44 电路的连续输出，这样的 $y(t)$ 利用起来会非常不方便。因此，D-A 转换器通常还扩展出了“零阶保持”功能（“Zero-order Hold”Functionality）。这意味着转换器将在下一个值得到转换前，一直保持着前一个值。事实上，在图 3.44 所示的电路中，如果在下一个离散时间点之前不去改变开关的位置，则它就能实现对前一次转换的保持。因此，转换器的输出是相应于序列 $y(t)$ 的阶梯函数 $y'(t)$。$y'(t)$ 是连续时域中的函数。

在例子中，考虑根据式（3.3）得出的转换输出结果，假定每个极性的转换各有 8 个步骤。对于这个场景，由于 $y'(t)$ 更容易观察，于是在图 3.45 中使用 $y'(t)$ 替代 $y(t)$。

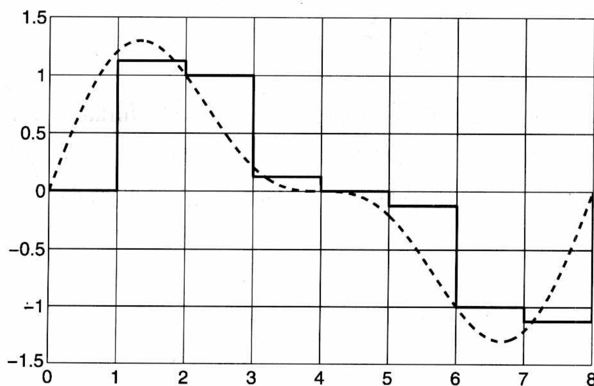



图 3.45　根据在整数时间点采样的 $e_3(t)$[式（3.3）]产生的阶梯函数 $y'(t)$

D-A 转换器允许将在时域上与数值均离散的信号，转换为在时域上与数值均连续的信号。但是，无论是 $y(t)$ 还是 $y'(t)$，它们都不能反映出被采样信号的原始

情况。

### 3.6.2 采样定理

假定在硬件回路中使用的处理器会将来自 A-D 转换器的值不加改变地送给 D-A 转换器，期待着将信号 $x(t)$ 存储在一张 CD 上，而后由它来产生完美的模拟音频信号。那么，有没有可能通过 D-A 转换器来重构出原始的模拟信号电压 $e(t)$（见图 3.8、图 3.20、图 3.43）呢？很明显，如果发生了在采样章节描述的混叠现象[⊖]，则不可能重构出原始的信号。因此，规定采集速率大于给定输入信号中最大正弦波频率的两倍［采样定理，参考式 3.8］。是不是只要满足了采样定理，就可以重构出原始信号了呢？下面作更进一步的分析。

向 D-A 转换器输入一离散序列的数值时，它将产生一个连续的模拟值序列。在两个采样时刻之间，D-A 转换器并不会产生输出结果，简单的零阶保持功能（如果有）将仅会产生阶梯函数。这意味着为了重构信号 $e(t)$，就需要无限大的采样速率，因为只有这样，所有的瞬时值才能被捕捉到。但是，也有许多很好的插值算法，可以在当前的采样值及前一次采样值之间插入更多数值，从而使输出更逼近原始函数。采样定律［Oppenheim et al.，2009］告诉我们，一个连续信号 $z(t)$ 可以从模拟数值序列 $y(t)$ 进行重构。

假定 $\{t_s\}$，$s=\cdots,-1,0,1,2,\cdots$ 是对输入信号的采样时刻集合，采样频率为常数 $f_s=\frac{1}{p_s}(\forall s:\ p_s=t_{s+1}-t_s)$。根据采样定律，可以从 $y(t)$ 中来估计出 $e(t)$，如下所示：

$$z(t)=\sum_{s=-\infty}^{\infty}\frac{y(t_s)\sin\frac{\pi}{p_s}(t-t_s)}{\frac{\pi}{p_s}(t-t_s)} \tag{3.24}$$

上式即广为人知的**香农—惠特克插值（Shannon-Whittaker Interpolation）定理**。$y(t_s)$ 是信号 $y$ 在采样时刻 $t_s$ 的采样值。随着后期的采样时刻 $t$ 远离采样时刻 $t_s$，$y(t_s)$ 对 $t_s$ 时刻的采样值影响越来越小。这种影响的降低遵循一个加权因子，即人们所知的 sinc 函数：

$$\mathrm{sinc}(t-t_s)=\frac{\sin\left(\frac{\pi}{p_s}(t-t_s)\right)}{\frac{\pi}{p_s}(t-t_s)} \tag{3.25}$$

上面的函数与 $|t-t_s|$ 一样，是单调递减的。加权因子用于计算在两个采样点

---

⊖ 事实上，如果有关于信号的其他附加信息，则完整的重构信号也是可能的。例如，如果这是一种特定的信号类型。

之间的数值。图 3.46 展示了对于 $p_s=1$ 时的加权因子。

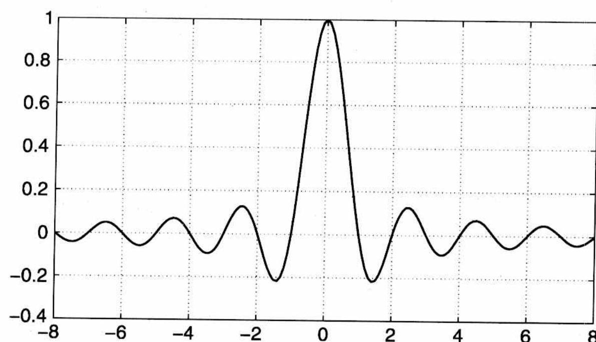



图 3.46　式（3.25）用于插值计算的视图

使用 sinc 函数，可以计算出式（3.24）中相应的和值。图 3.47 与图 3.48 展示了当 $e(t)=e_3(t)$ 时，使用相同的函数($x(t)=w(t)$)进行处理的结果。

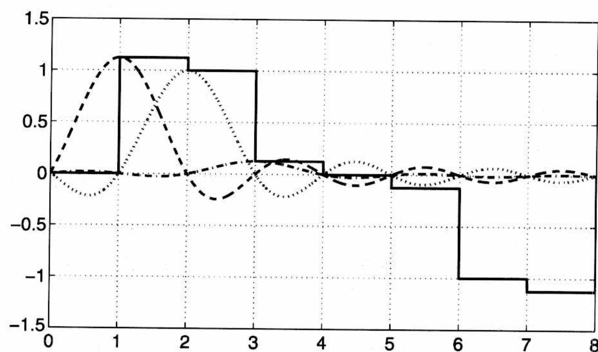



图 3.47　$y'(t)$（虚线）与式（3.24）的前三项结果

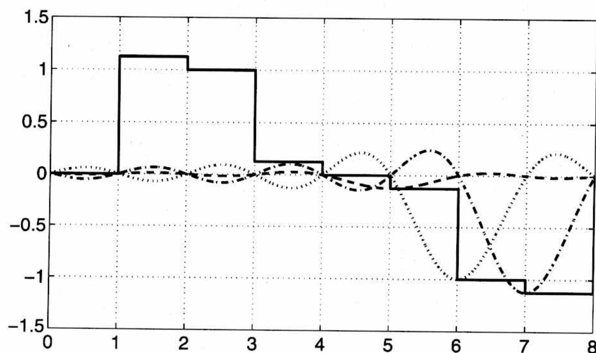



图 3.48　$y'(t)$（虚线）与式（3.24）的最后三项非零结果

在每个采样时刻 $t_s$（在这里的例子中只有使用整数时间），因为对于其他采样

时刻 sinc 均为零，于是由 $y(t_s)$ 可以计算出相应的 $z(t_s)$。在两个采样时刻之间，所有的邻近离散值都对结果 $z(t)$ 有影响。图 3.49 展示了当 $e(t)=e_3(t)$ 时，使用相同的函数（$x(t)=w(t)$）时的结果 $z(t)$。

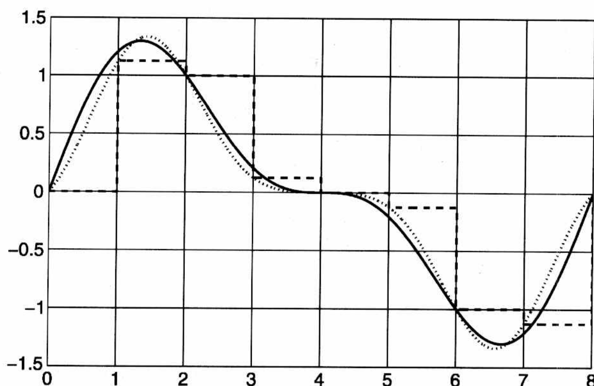


图 3.49　$e_3(t)$（实线）、$z(t)$（点线）以及 $y'(t)$（虚线）

图 3.49 中包含着信号 $e_3(t)$（实线），$z(t)$（点线），以及 $y'(t)$（虚线）。$z(t)$ 已经考虑了在图 3.47 以及图 3.48 中所有采样点的影响。$e_3(t)$ 与 $z(t)$ 非常相似。

那么，使用式（3.24）怎样才能得到与原始输入信号最相近的信号呢？采样定理告诉人们（参考如 [Oppenheim et al., 2009]），如果满足了采样定理 [式(3.8)]，则使用式（3.24）**可以计算出非常精确的估计值**。因此，将分析如何使用式（3.24）。

在电子系统中如何计算式（3.24）呢？由于式（3.24）将产生连续信号，因此使用数字信号处理器并不能在离散系统中对它进行计算。使用模拟电路来对此复杂的公式进行计算，看起来也是非常困难的事情。

幸运的是，所要进行的计算是在信号 $y(t)$ 与 sinc 函数之间的操作，也被称为折叠操作（Folding Operation）。根据经典的傅里叶变换理论，时域中的折叠操作与频域中基于频率的滤波函数的乘法是等价的。这里的滤波函数是时域中相应函数的傅里叶变换。因此，可以使用一些适当的滤波器来对式（3.24）进行计算。图 3.50 展示了滤波器的相应位置。

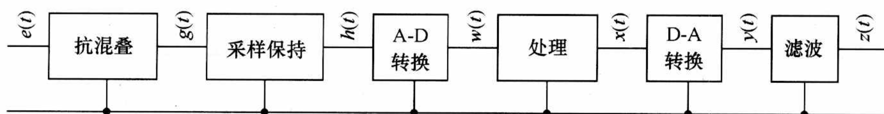



图 3.50　信号 $e(t)$ 转换为模拟量再到数字量再转换为模拟量

那么剩下的问题是哪一种基于频率的滤波函数是 sinc 函数的傅里叶变换？使

用一个低通滤波器来计算 sinc 函数的傅里叶变换［Oppenheim et al.，2009］。因此，为了计算式（3.24），要做的“全部”工作就是让信号 $y(t)$ 通过一个低通滤波器，在图 3.51 中展示了“理想滤波器”的滤波频率。函数 $y(t)$ 作为多个正弦波的和，即使假如在输入端加上了抗混叠滤波器，它仍将含有部分高频成分，滤波器总是具有冗余性。

还有一个问题：理想的低通滤波器是不存在的。因此，必须根据滤波器的特性进行折中考虑。事实上，需要许多仍然有缺陷的方式来保持最大限度的可以对输入信号进行重构：

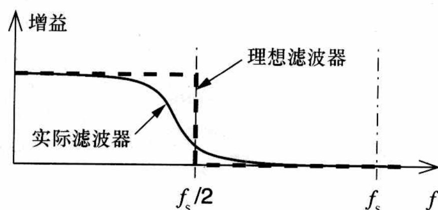



图 3.51　理想低通滤波器（虚线）与实践中的低通滤波器（实线）

1）理想的低通滤波器不可能被设计出来，因此必须对这类滤波器使用近似法。如何在设计时进行折中与平衡是一门艺术（这在许多领域被使用，如音频设备）。

2）基于同样的理由，不可能移除输入频率中超出奈奎斯特（Nyquist）频率成分的部分。

3）图 3.49 展示了数值均衡的影响。正是由于数值均衡，$e_3$（$t$）在很多时候与 $z$（$t$）不一样。由 A-D 转换器引入的量化噪声，在产生输出时并不能被移除。A-D 转换器的输出信号 $w$（$t$）仍将受到量化噪声的影响。但是，量化噪声并不会影响到来自采样保持电路的输出信号 $h$（$t$）。

4）式（3.24）基于多个在将来的采样时刻的数值和。在现实中，可以将信号扩展到有限个“将来的”采样点。无限的延时是不可能得到的。在图 3.49 中，就并没有考虑图形之外采样点的影响。

低通滤波器的功能是由模拟电路实现的：由于离散时间与数值的固有限制，在数字域是不能实现这种模拟滤波器的功能的。

许多的学者都在采样理论方面有所研究，因此有许多人与采样定理相关。这些学者，如 Shannon、Whittaker、Kotelnikov、Nyquist、Küpfmüller 等。因为无法将如此多的人与一个定理都联系起来，因此把将信号进行重构的理论统一称为采样定理。

### 3.6.3　执行器

现实中有大量的执行器［Elsevier B. V.，2010a］，从大到可以搬运数吨重量的执行器，小到集成在 μm 面积内的执行器，如图 3.52 所示。

对于执行器进行全面介绍比较困难。此处来看在未来会越来越重要的一种特殊执行器：微系统技术使更小的执行器制造成为可能，而这种微小的执行器甚至可以被植入到人体内。

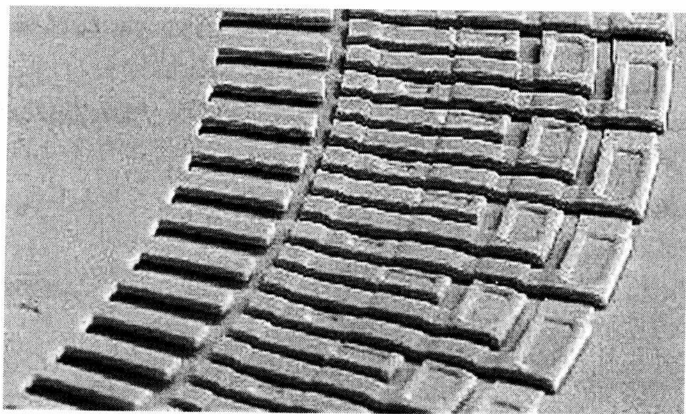


图 3.52　基于微系统技术的执行器电动机
[（Courtesy E. Obermeier，MAT，TU Berlin），© TU Berlin]

使用这种微小的执行器，药物可以根据人体的病因进行相应的动作，提供更好的诊断与治疗方式。图 3.52 展示了使用微系统制造的一个微小的电动机，它集成在一块 μm 级的区域上，它的转动机构由静电来进行驱动。

## 3.7　安全硬件

嵌入式系统通常可能还有安全方面的设计需求。如果安全是产品中的一个重要因素，则可能要有针对性的相应开发硬件。在通信与存储中，安全也是必须被保证的因素 [Krhovjak and Matyas，2006]。同样，密钥的产生也需要特定的安全设备。当前业界已经设计了许多硬件安全组件，这些组件的设计目的，是为了防御诸如测试供电电流或者电磁辐射从而获取信息，这种攻击方式被称为边信道攻击（Side-channel Attacks）。这些组件包含一些物理介质的保护方式（如屏蔽、加扰等）。有一些处理器也支持加密与解密。在物理安全保护之外，还需要一些软件安全保护，当前比较典型的是使用密码学方法。智能卡就是一种常见的安全设备，它必须运行一些加密算法来保证它的运行安全。通常，区分不同等级的安全，以及区分“入侵者”的等级，在安全系统中都非常重要。关于安全硬件的设计技术已经超出了本书的讨论范围。有兴趣的读者可以参考 Gebotys [Gebotys，2010] 的著作，以及部分研讨会文章 [Clavier and Gaj，2009]。

## 3.8　思考题

1. 建议读者使用自己的小机器人来对图 3.14 中展示的硬件回路进行演示。机器人应当包含传感器与执行器，它的控制程序实现了完整的控制回路。如机器人使用了光学传感器，这样它就可以沿着地板上一条黑色的线行进。具体的实验与步骤

依赖于读者所使用的机器人。

2. 为什么嵌入式系统的优化非常重要？比较一下嵌入式系统中信息处理能耗方面的相关技术。

3. 假定有一个输入信号 $x$，它是 1.75kHz 与 2kHz 两个正弦波信号的和。以 3kHz 的采样速率对 $x$ 进行采样，多个时间点采样后，有可能对原始信号进行重构吗？解释一下你的答案。

4. 从 A-D 转换器产生离散值：绘制基于闪存 A-D 转换器的电路，它的输入可以是双极性信号，其输出至少能区分 8 个电压梯度。

5. 比较一下基闪存 A-D 转换器与连续比较型 A-D 转换器。假定想区分 $n$ 个电压梯度，使用 O 标志（O-nation）在图 3.53 中填写它们的复杂度。

| | 基于闪存的转换器 | 连续估值转换器 |
|---|---|---|
| 时间复杂度 | | |
| 空间复杂度 | | |

图 3.53　A-D 转换器的复杂度

6. 假定使用一个 4bit 的逐次逼近型 A-D 转换器。输入电压的范围从 $V_{min}$ = 1 V（ = "0000"）到 $V_{max}$ = 4.75 V（ = "1111"）。对于 2.25V、3.75V 与 1.8V 的电压，各需要多少次转换？用一个与图 3.12 类似的图形来说明 A-D 转换器对这些电压是如何进行逐次逼近的。

7. 对基于闪存 A-D 转换器进行扩展，使其可以转换双极性的电压信号。

8. 假定在思考题 4. 中，使用一个正弦波作为输入信号，描述在这种情况下的均衡噪声信号。

9. 列出 DSP 的特性。

10. FPGA 包含着哪些模块？在实现布尔运算时，会用到 FPGA 的哪些模块？FPGA 是如何进行配置的？它是一种高能效的器件吗？FPGA 适合哪些应用场景？

11. 在关于内存的讲述中，有时说“小即优秀”，这样说的理由是什么？

12. 开发这样的一个 FlexRay™ 簇：这个簇包含了 $A$、$B$、$C$、$D$ 和 $E$ 5 个节点，所有节点通过两个通道互连。簇使用总线拓扑，由于节点 $A$、$B$、$C$ 执行着安全关键的任务，因此它们在总线上的请求必须在 20 宏拍（macroticks）内得到回应。下面是希望读者去完成的部分：

1）下载 levi FlexRay 模拟器 [Sirocic and Marwedel, 2007a]，解压 .zip 文件并安装它。

2）执行 leviFRP. jar 从而进行训练模式。

3）在训练模式下设计上面描述的 FlexRay 簇。

4）配置通信周期，保证节点 $A$、$B$、$C$ 在总线上的最大访问延迟在 20 宏拍以内，节点 $D$、$E$ 仅使用通信周期中的动态段。

5）配置节点的总线请求。节点 $A$ 在每个周期都发送一条消息，节点 $B$ 与 $C$ 每两个周期发送一条消息，节点 $D$ 在每个周期发送一条长度为两个微时隙（minislots）的消息，节点 $E$ 每两个周期发送一条长度为两个微时隙的消息。

6）在可视化视图中检查节点 $A$、$B$、$C$ 的总线请求能否被保证得到响应。

7）在动态段中交换节点 $D$ 与节点 $E$ 的位置，会导致什么结果？

13. 设计一个 3bit D-A 转换器的电路图，转换器需要对一个 3bit 的向量 $\boldsymbol{x}$ 完成编码。证明输出电压与输入向量 $\boldsymbol{x}$ 之间是成比例关系的。如果 $\boldsymbol{x}$ 表示的是两个互补的数，电路应当如何修改？

14. 附录 B 中的图 B. 4 是一个放大器，如果在输入端加上电压 $V_1$，则有

$$V_{out} = g_{closed} \cdot V_1$$

计算以 $R$ 和 $R_1$ 表达的图 B. 4 中的增益 $g_{closed}$。

# 第4章　系统软件

并不是所有的嵌入式系统的设计都要从头做起，有一些符合标准的模块可以实现复用，这些模块由一些之前设计的成果以及某些含有**知识产权**的模块构成。在日渐复杂的系统中，对含有知识产权的模块进行复用，是加快系统设计速度的一种积极应对的方式。术语“知识产权模块复用”更多的是指对于硬件模块的重用，但是仅仅对硬件进行重用还远远不够。Sangiovanni-Vincentelli 指出，软件也应该像硬件一样进行复用。因此，Sangiovanni-Vincentelli [Sangiovanni-Vincentelli，2002] 倡导在设计阶段就应该同时考虑到软、硬件的复用性。

系统软件模块作为标准软件模块的一部分是可以进行复用的，这其中包括嵌入式操作系统以及**中间件**。中间件是指软件在操作系统以及应用软件之间提供了一个中间层。将库文件看作一种特殊的，用于通信的中间件，操作系统提供这些库文件作为基本的通信工具。并且，认为实时的数据库是第二层次的中间件（参见 4.5 节）。调用标准的软件模块可能已经包含在了规格说明书中。有关这些标准模块的 API（Application Programming Interface，应用程序接口）信息可能需要完全执行 SUD 的规范。

参照设计流程，将会在本章进行嵌入式操作系统以及中间件的描述（见图 4.1）.

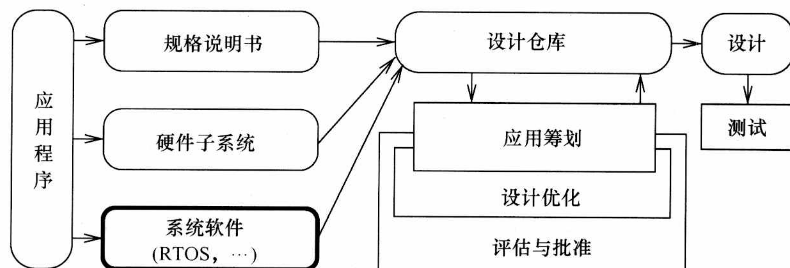



图 4.1　简化的设计流程图

## 4.1　嵌入式操作系统

### 4.1.1　总体需求

除了极其简易的系统外，调度系统、进程切换以及 I/O 管理都需要一个适合的

嵌入式操作系统作为支撑。进程调度（或者叫做进程分配）可以使每一个任务都像自己有一个独占的处理器一样运行。

通过虚拟内存系统，可以使不同的进程以及线程运行在不同的地址空间上。**每个进程都拥有自己的地址空间，反之若干线程有可能共享同一段地址空间。**相对于不使用虚拟内存系统的系统，使用虚拟内存的系统在进行进程的上下文切换时，需要花费更多的时间。线程的特点就是可以共享同一段地址空间，并可以使用共享内存进行线程间通信。操作系统也必须向进程和线程提供同步及通信的方法。更多的有关系统软件的知识可以在操作系统类的相关教科书上获得，例如 Tanenbaum [Tanenbaum, 2001]㊀所著书籍。

下面是嵌入式操作系统一些必不可少的功能：

1）鉴于嵌入式系统需求的多样性，嵌入式操作系统所能提供的功能也是多种多样的。由于对系统性能要求比较高，运行操作系统所能提供的所有功能几乎是不可能的。对于绝大多数应用程序来说，操作系统必须尽可能短小精悍，因此需要针对所要运行的程序对操作系统进行**灵活定制**。**可配置性**也是嵌入式操作系统非常重要的特性之一。在实现操作系统的可配置性上，也有着丰富的手段，这包括㊁：

① 面向对象，可以用来衍生出适合的子类：例如，可以通过一个通用的调度器的类，从这个类中，可以衍生出适当的、具有自己特点的子类。然而，面向对象的方法通常会产生一些额外的系统开销，例如面向对象特征中的多态性，就会产生额外的运行时开销。这些开销对于对性能要求较高的系统来说，有时显得难以接受。

② 面向方面的编程 [Lohmann et al., 2009]：随着这个概念的不断研究和发展，正交软件架构可以描述为一种，子模块具有较强独立性以及相关模块可以无需考虑其他模块的相关性即可添加的编码过程。例如，对某些代码的分析可以被描述为一个单独的模型。这些代码可以不经思索从相关的源码中进行添加和删除。CIAO 系列的操作系统即采用了这种方法进行设计 [Lohmann et al., 2006]。

③ 条件编译：通过使用#if 以及 #ifdef 预处理命令可以使用宏进行预处理以实现条件编译。

④ 高级编译时评估，在编译操作系统之前，可以将一些变量预先赋予一些常量的值，编译器尽可能将这些值向下进行传递，这将会对编译器的性能产生一定的优化。假如某个特定的函数参数一直是常量，那么这个参数就可以从该函数的参数列表中移除。局部评估 [Jones, 1996] 为上述的编译器优化提供了一个框架。在某些情况下，动态的数据有可能被静态数据替换掉 [Atienza et al., 2007]。一项有关操作系统定制的报告已由 McNamee [McNamee et al., 2001] 等人发表。

---

㊀ 如果本书读者没有学习过有关操作系统的相关课程，那么在进行后续部分知识的学习前，有必要进行一下操作系统相关知识的学习。

㊁ 此列表由开发过程中的开发顺序或者工具链作为排序标准。

⑤ 无用函数链接时移除，在目标链接时的早期，会有很多使用或未使用的函数的信息。例如，链接器会指出究竟使用了库里的哪些函数，至于没有使用到的函数则可以被丢弃掉［Chanet et al.，2007］。

这些技术往往被包含于操作系统的某些可选择的规则文件中。裁剪操作系统时，可以通过图形用户界面进行功能配置，从而使得裁剪变得更加容易。例如，Wind River 公司的 VxWorks［Wind River，2010a］就可以通过图形界面进行配置。

对于裁剪出来的操作系统而言，功能验证也是一个潜在的问题。每一个经裁剪得出的操作系统都必须经过充分的验证。Takada 提到，对于 eCos 来说，如何验证大约 200 个配置选项是一个潜在的问题（eCos 是一个由 Red Hat 推出的开源的实时操作系统［Massa，2002］）。产品线软件工程［Pohl et al.，2005］针对上述问题提出了若干解决方案。

2）嵌入式系统中有大量的、多种多样的外围设备。许多嵌入式系统并没有硬盘、键盘、屏幕或者鼠标。也许对于操作系统来说，除了系统时钟外，没有什么设备是必要的。应用程序经常被设计用来处理各种各样特殊的设备。在这种情况下，设备不会被应用程序共享，并且操作系统也没有管理设备的必要。由于设备的数量确实庞大，操作系统也难以将所有能支持的设备驱动都放入操作系统中。因此，将操作系统和驱动进行分离，仅通过一个特殊的任务进行设备管理就比将驱动集成进操作系统有必要得多。并且由于大量的嵌入式设备的速度较低，从性能角度出发考虑，也没有必要将其集成进操作系统中。由此则会产生不同的软件层。就 PC 来说，像硬盘驱动、网络驱动以及音频驱动都被假定为是存在的。这些驱动都在较低的软件层中实现。作为所有应用程序的标准，API 和中间件在顶端实现，对于嵌入式操作系统来说，设备驱动的实现是在内核之上的。应用程序和中间件有可能是在驱动之上实现，而不是在操作系统的 API 上实现（见图 4.2）。

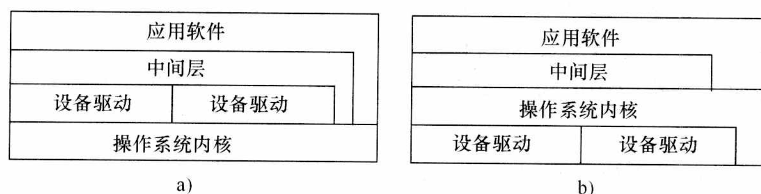



图 4.2　在操作系统内核之上（图 4.2a）和之下（图 4.2b）实现的设备驱动

此处再次使用 VxWorks 为例。图 4.3 显示了一部分 Wind River® 公司的平台架构［Wind River，2010a］。

3）**保护机制并不总是必要的**，由于嵌入式系统经常被设计用于某种单一的目的（他们并不被认为用于支持所谓的多道程序），因此未经验证的程序几乎不可能被载入运行。当软件经过测试后，该软件就被认定为是可靠的，这也适用于 I/O 中。与桌面程序形成鲜明对照的是，它们没有实现 I/O 指令作为特权指令的需要，

任务允许去做它们自己的I/O操作。与之前相比，这样可以很好地减少I/O操作的系统开销。

<table>
<tr><td>…</td><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>CAN</td><td colspan="3">TCP/IP</td><td>闪存文件系统</td></tr>
<tr><td>串口</td><td>PPP</td><td>以太网</td><td>USB</td><td>DOS文件系统</td></tr>
<tr><td colspan="5">VxWorks实时操作系统</td></tr>
<tr><td colspan="5">BSP开发工具</td></tr>
<tr><td colspan="5">硬件文档以及相关工具</td></tr>
</table>

图4.3　Wind River® 公司开发平台软件层次

**示例**：switch对应需要进行I/O地址映射的开关，这些开关需要某些程序来进行检测。可以使用指令

```
load register,switch
```

去检查这些开关。这种操作没有通过使用操作系统调用的必要，因为系统调用会造成频繁的系统上下文切换（寄存器组等），导致不必要的系统开销。

在对灵活性要求较高的嵌入式系统中，不使用保护机制是一种趋势，但是基于安全性的需求也使得保护机制变得非常必要。特殊存储器管理单元就是在这种情况下被提出的（参见Fiorin [Fiorin et al.，2007] 相关著作）。

4）**中断可以与任何进程相关联**。通过操作系统的系统调用，当中断发生时，可以要求操作系统开始运行或者停止运行任务。甚至可以把任务的起始地址保存在中断向量表中，但这样做非常危险，因为操作系统有可能忽视实际运行的任务。并且，可组合性可能也要受到这些影响，假如一个特殊的任务直接关联到了某些中断上，那么这个任务可能很难添加到其他任务中，因为其他任务也需要一些事件才能够触发并启动。特殊应用的设备驱动（假如被用到）也许会在中断和任务之间建立链接。

5）许多嵌入式系统是实时系统，因此，这些系统中的操作系统必须是实时操作系统。

有关嵌入式操作系统的更多内容可以在Bertolotti [Bertolotti，2006] 所著书中的有关章节获得。该书中包含了嵌入系统的架构、POSIX标准、开源实时操作系统以及一些虚拟化方面的内容。

## 4.1.2　实时操作系统

**定义**：实时操作系统是用来支持实时系统的操作系统 [Takada，2001]。

那么如何认为一个操作系统是否为实时操作系统呢？有如下4个关键点用于

判断：

1）**操作系统的时间行为必须可预测**。对操作系统的任何服务来说，都必须有一个明确的执行时间的上界作为保证。实际上，这里也有不同的针对时间的预测等级。比方说，对于某组系统调用来说，明确地知道这组系统调用的运行时间的上界，而另一种系统调用的运行时间则较不确定。类似于“获得当前时间”的系统调用就属于此类。而另一种调用，则可能有较大的变化。例如分配 4MB 空闲内存的调用，则可能属于第二类。特别要提及的，实时操作系统的调度策略必须具有确定性。

中断在某些时刻需要被禁止，以避免系统之间各个模块互相干扰。更有甚时，为了避免任务之间的互相影响，中断也应该被禁止。关闭中断的时间必须尽可能短，以避免一些不可预测的系统延迟以及某些关键时间的处理。

由于实时操作系统的文件系统有时需要在硬盘上部署，所以应该尽可能将连续的文件（文件应存储于连续的磁盘空间上）放置在连续的柱面，以避免磁头进行无法预测以及没有必要的移动。

2）**操作系统必须进行任务调度管理**。多个任务的执行时间间隔，以及特定场景下的初始执行时间，即调度。而且，操作系统应该知道每一个任务的死线，以便于操作系统应该申请恰当的调度策略（操作系统只需要提供在指定时间或者指定优先级启动任务的服务）。有关调度算法的细节会在第 6 章进行讨论。

3）**有些系统也需要操作系统进行时间管理**。如果进程与物理环境相关联，那么操作系统就必须要进行时间管理。物理时间由实数来描述，在计算机中常被离散时间代替。根据具体的需求有可能不同：

① 在有些系统中，与全球标准时间进行同步是必需的。在这种情况下，**全球时钟同步**概念被提出。有两个标准可以用来进行时间同步：

a. 协调世界时（Universal Time Coordinated，UTC）：UTC 由天文学上的标准来进行定义。由于地球运动的变化，这个时间标准必须要随时进行调整。在每一年的年终交替时，有可能需要添加若干秒的时间。这个调整有可能会造成一些问题，由于有些软件设计的缺陷，在跨年时有可能会两次进入到下一年。

b. 国际原子时间［通过原子钟进行定时，法语为 Temps Atomic Internationale.（TAI）］，这个标准不受任何人为影响。

有时与外部环境关联的目的就是为了获取准确的时间。外部的同步通常基于无线通信协议，例如全球定位系统（GPS）［National Space-Based Positioning，Navigation，and Timing Coordination Office，2010］或移动电话网络。

② 如果嵌入式系统用于网络，那么它通常可以从网络系统中获取充足的同步时间信息与本地时间进行同步。假若这样，互连的嵌入式系统就可以尝试取得当前时间的一致视图。

③ 在有需要的前提下，提供尽可能精准的本地时延。

对有些应用来说，提供高精度的时间服务是充分条件。举例来说，有时这些应用需要分辨错误发生的先后顺序。例如，它们可以帮助鉴别导致停电的原因是否是由发电装置引起的（参见［Novosel，2009］）。时间服务的准确度取决于特定的执行平台的支持。如果时间服务程序在进程级别中实现，那么时间精度（微秒级别）将变得极不准确；如果时间服务程序在硬件级别提供，那么时间精度（微秒级别）将变得非常准确。更多有关时间服务以及时钟同步的信息，请参考 Kopetz［Kopetz，1997］的相关书籍。

4）**操作系统必须极有效率。**如果操作系统的效率不够，那么上面所有提到的内容都是无用的。因此，操作系统必须足够快。

每个实时操作系统都包括一个所谓的实时操作系统内核。这些内核所管理的资源可以在任何一个实时操作系统中找到，包括处理器、内存以及系统时钟。内核的主要工作包括任务管理、作业间同步以及通信、时钟管理和存储器管理。

有些实时操作系统被设计用来做通用的嵌入式应用，有一些则专注于某些特殊领域。例如，OSEK/VDX 操作系统主要用于汽车控制领域。操作系统对于指定的某些领域可以提供一些专属的服务，这样的操作系统对于这样的领域结合更加紧密，在这个领域上也往往优于能够提供多种服务领域的操作系统。

同样地，一些 RTOS 提供标准的 API，而另一些 RTOS 会提供它们自己专属的 API。例如，有些 RTOS 遵从了 UNIX POSIX RT-extension 的接口标准［Harbour，1993］，有些则使用的是 OSEK/VDX 标准，有些则使用了日本的开发标准 ITRON。许多实时内核都有自己的 API，上文提到过的 ITRON，是一个使用链接时配置的成熟的 RTOS。

可用的 RTOS 可以被进一步分为如下三类［Gupta，2002］：

① **快速专用内核。**Gupta 说："对于复杂系统来说，系统运行的速度是重中之重。这些内核的设计基点就是尽可能快的运行，而不是进行所有事情的预测。"这种系统包括 QNX、PDOS、VCOS、VTRX32、VxWorks。

② **对标准操作系统的实时性扩展。**为了尽可能利用和适配主流的操作系统而又同时支持实时性，一种既支持实时任务也支持普通任务的混合系统得到了充分的发展。在这类系统中，有一个实时的内核执行所有实时的任务。另外有一个标准的操作系统去执行所有非实时的任务（见图 4.4）。

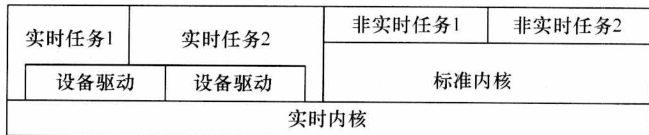



图 4.4　混合操作系统

这种混合式操作系统有一些优点：系统可以使用标准的操作系统的 API，可以

拥有GUI（Graphical User Interfaces，图形用户接口）、文件系统等。而且可以使得标准的操作系统更快地适应嵌入式引用。另外，这些标准操作系统的非实时任务也不会对实时任务产生任何消极的影响，甚至标准操作系统的崩溃也不会影响到这些实时任务。不利的一面是（这些也可以从图4.4中见到），在设备驱动方面可能会遇到一些问题，因为标准操作系统也会拥有自己的设备驱动。为了避免实时任务驱动以及其他任务驱动之间的影响，实时任务和非实时任务有可能需要对设备分开进行操作。虽然也有很多的尝试在尽可能不影响实时性能的前途下去试图弥补这两者之间的鸿沟，但是对于实时任务来说，很多标准操作系统能提供的服务，例如文件系统访问以及GUI还是不能被实时任务所使用。RT-Linux是这种混合类操作系统的一个示例。

Gupta [Gupta，2002] 说到，尝试使用某个版本的标准操作系统“是一种不正确的方法，因为有太多的基本的和不恰当的潜在假设仍然存在，例如对普通用例的优化（不是对最坏的用例），忽略大多数的语义信息（如果不是全部的语义信息）以及独立的CPU调度和资源分配”。的确，大多数标准操作系统的应用之间的任务依赖并不是非常频繁，因此经常会被这样的系统忽略。这种情况不同于嵌入式系统，由于任务间的依赖相当普遍，这些情况也应当被考虑进去。不幸的是，基于标准操作系统扩展的实时系统很难对这些情况进行处理，此外资源的分配以及调度也很少与标准的操作系统相结合。然而，为了确保对时间的限制要求，资源的分配以及调度算法集成还是必不可少的。

③ 有很多的**研究体系**把目光瞄向了上面的那些限制，这其中包括Melody [Wedde and Lind，1998] 以及MARS（基于Gupta的研究 [Gupta，2002]）、Spring、MARUTI、Arts、Hartos和DARK。

Takada [Takada，2001] 提出了一种低开销的内存保护机制，暂时地保护计算所用到的资源（针对预防任务使用比一开始分配的时间更多的时间），用于片上多核系统（特别是异构多处理器以及多核处理器）的RTOS以及用于连续的媒体流数据处理以及服务控制的保证。

基于嵌入式系统市场增长的潜力，标准操作系统的开发商开始积极地尝试去销售他们产品的一些变化版本（例如嵌入式Windows [Microsoft Inc.，2003]），并且开始尝试从一些传统的嵌入式厂商（例如Wind River）手中获取市场份额。

### 4.1.3　虚拟机

在有些环境中，使用一个真实的处理器去模拟许多处理器是非常有用的。通过**虚拟机**在硬件上的执行，使得这一想法成真。在虚拟机上，可以运行多种不同的操作系统。很明显，这允许多个操作系统同时运行在一个处理器上。在嵌入式系统上，虚拟机务必要小心地使用，因为系统的暂态行为也有可能导致产生很多的问题，而且时间的可预测性也有可能会丢失。但是虚拟机在某些情况下也有其独到的

用处。例如，有可能需要在一个处理器上运行多个操作系统遗留的应用。虚拟机的所有功能已经超出了本书所能描述的范围。感兴趣的读者可以去参考 Smith 等人［Smith and Nair，2005］所著的相关书籍以及 Craig［Craig，2006］所著的相关书籍。PikeOS 是一个致力于虚拟化方面的嵌入式操作系统的范例［SYSGO AG，2010］。PikeOS 允许系统资源（例如存储器、I/O 设备、CPU 时间片）划分为各个独立的子集。PikeOS 使用的是微内核，许多操作系统的 API 以及实时运行环境（RTE）都可以在这个内核的基础上运行（见图 4.5）。

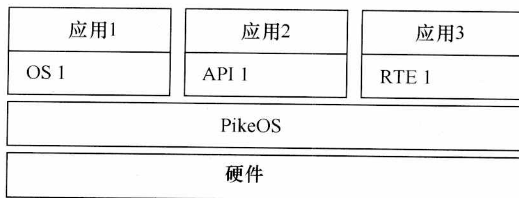


图 4.5 PikeOS 虚拟化（© SYSGO）

## 4.1.4 资源访问协议

### 4.1.4.1 优先级反转

在某些情况下，任务必须互斥地进行资源访问。例如全局共享变量或者是某些设备，以避免不确定或者不期望的进程的行为。对嵌入式系统来说，这种互斥操作非常重要，例如基于共享内存的进程间通信或者互斥地去访问某些特定的硬件设备。需要互斥访问这些共享资源的代码段称之为**临界区**，被临界区访问的资源称之为**临界资源**。临界区应尽可能短，操作系统特别提供了一系列的原语用来申请以及释放访问这些临界资源，这种原语称作**互斥锁**。直到被持有的锁被释放，临界区才被允许去访问临界资源。因此，没有获得锁的临界区必须去等待锁被释放，并且恢复优先级最高的进程得到执行。

本书中，将请求锁得操作命名为 P（S）操作，并且将释放锁的操作命名为 V（S）操作，S 表示所要请求和释放的锁。P（S）操作以及 V（S）操作又被统称为**信号量操作**。信号量允许最多 $n$ 个（$n$ 表示一个变量）线程或者进程同时访问某段临界资源。S 是一个可维护的数据结构已表示还有多少临界资源可用。P（S）操作检查计数，当没有可以使用的临界资源时，阻塞调用者。如果有可用临界资源，则修改计数，并且调用者也可继续执行。V（S）增加可用资源的数量，并确保假如有被阻塞的临界区时，解除对临界区的阻塞。P（S）操作以及 V（S）操作源自于荷兰语，将使用这些操作当互斥的信号量，也就是 $n=1$ 时的信号量。当信号量作为互斥量时，同一时刻只允许一个临界区去访问临界资源。

对嵌入式系统来说，任务之间经常是有依赖关系的，而非相互独立。另外，实时应用的优先级也比非实时任务的优先级更加重要。在这种情况下，对临界资源的访问有可能导致优先级反转，优先级反转即低优先级的任务有可能在高优先级的任务执行之前先被执行。优先级反转在非嵌入式系统中也存在。然而，由于之前列出

的原因，可以认为在嵌入式系统中发生优先级反转的情况更加让人无法接受。

第一个例子是由“临界资源”以及“不支持抢占”导致的优先级反转，如图 4.6 所示。

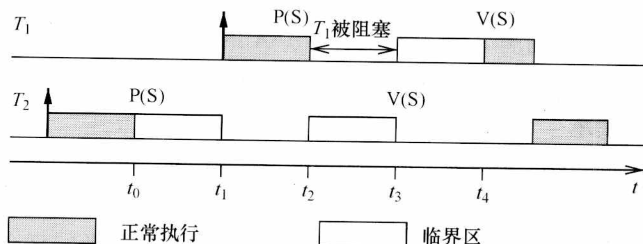



图 4.6　被低优先级的任务阻塞

**黑色的向上指向的箭头**表示在这一时刻哪个任务开始执行，或者是进入“就绪态”。在 $t_0$ 时刻，任务 $T_2$ 进入临界区后，要求独占地访问临界资源，并使用了 P 操作。在 $t_1$ 时刻，任务 $T_1$ 进入就绪态，并抢占了任务 $T_2$ 的运行。在 $t_2$ 时刻，由于 $T_2$ 持有临界资源的互斥锁，$T_1$ 的临界区试图获取临界资源时，被阻塞掉。任务 $T_2$ 恢复执行并在某一时刻释放掉了临界资源。释放的操作会去检查被挂起的任务是否有优先级高于 $T_2$ 的，如果有，那么抢占 $T_2$ 的处理器资源。此时从 $T_1$ 的阻塞可以看到，一个低优先级的任务阻塞掉了一个高优先级的任务。需要互斥的访问某些临界资源是导致这种结果的主要原因。幸运的是，图 4.6 只是某些特殊情况，阻塞的持续时间不会超过 $T_2$ 的临界区长度。这种情况会导致系统发生异常，并很难避免。

在有些情况下，事情还会变得更糟。可以在图 4.7 中看到相关的例子。

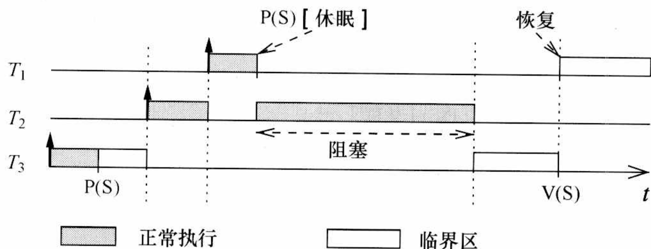



图 4.7　潜在的长时间优先级反转

假定有 $T_1$、$T_2$ 以及 $T_3$3 个进程。$T_1$ 的优先级最高，$T_2$ 的优先级居中，$T_3$ 的优先级最低。另外，假定 $T_1$ 和 $T_3$ 需要通过 P 操作互斥地访问某些临界资源。现在，当 $T_3$ 对临界资源进行访问时，被 $T_2$ 抢占。当 $T_1$ 抢占 $T_2$ 并尝试去访问 $T_3$ 正在占据的临界资源时，由于没有获得相应的互斥锁导致 $T_1$ 被阻塞掉，此时 $T_2$ 开始继续运行。一直到 $T_2$ 运行结束之前，$T_3$ 都不能释放临界资源。尽管 $T_1$ 的优先级高于 $T_2$，但 $T_1$ 还是被优先级比自己低的 $T_2$ 阻塞了。在这个例子中，$T_1$ 直到 $T_2$ 运行结束才

可以运行。$T_1$ 被低于自己优先级并且不在它的临界区的任务阻塞。这就叫做**优先级反转**[⊖]。实际上，尽管 $T_2$ 与 $T_1$ 和 $T_3$ 无关，但优先级反转还是发生了。优先级反转的持续时间并不是以临界区的长度作为边界的。这些例子都可以用 Levi 仿真软件进行仿真［Sirocic and Marwedel，2007c］。

另一个优先级反转的著名例子发生在 NASA 的火星探路者上，当火星探路者正在火星上执行任务时，共享内存导致了一个优先级的反转［Jones，1997］。

#### 4.1.4.2 优先级继承

优先级继承是解决优先级反转的一个好办法。这是一个可以在绝大多数实时操作系统中都可以使用的、标准的解决方法。解决流程如下：

1）通过任务的优先级进行任务的调度。具有相同优先级的任务遵从先到者先服务的原则。

2）当任务 $T_1$ 执行 P 操作并且临界资源已经被任务 $T_2$ 获得时，$T_1$ 就会被阻塞住。如果 $T_2$ 的优先级低于 $T_1$，那么 $T_2$ 继承 $T_1$ 的优先级，因此 $T_2$ 恢复执行。总之，每个任务都继承被自己阻塞的、最高优先级任务的优先级。

3）当任务 $T_2$ 执行 V 操作时，它的优先级减少为仍被它阻塞的最高优先级的任务。假如没有任何任务被 $T_2$ 阻塞，那么它的优先级恢复成 $T_2$ 初始的优先级。此外，被互斥量阻塞的最高优先级的任务此时恢复执行。

4）优先级继承是可传递的：如果 $T_x$ 阻塞了 $T_y$ 并且 $T_y$ 阻塞了 $T_z$，那么 $T_x$ 继承 $T_z$ 的优先级。

这样，被低优先级任务阻塞的高优先级任务通过传递它们的优先级给低优先级的任务，就可以使得低优先级的任务尽可能快地释放信号量。

在图 4.7 所示的示例中，当 $T_1$ 执行 P 操作时，$T_3$ 将会继承 $T_1$ 的优先级。这将会避免之前提到过的 $T_2$ 抢占 $T_1$ 时导致优先级反转（见图 4.8）。

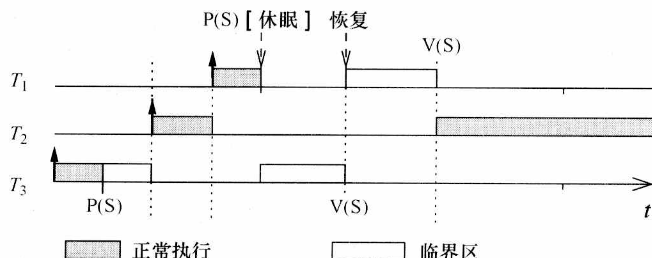



图 4.8　图 4.7 使用优先级继承

图 4.9 显示了临界区嵌套的例子［Buttazzo，2002］。

---

⊖ 有些作者确实认为图 4.6 可以被看作优先级反转。本书更早之前的版本也是这样认为的。

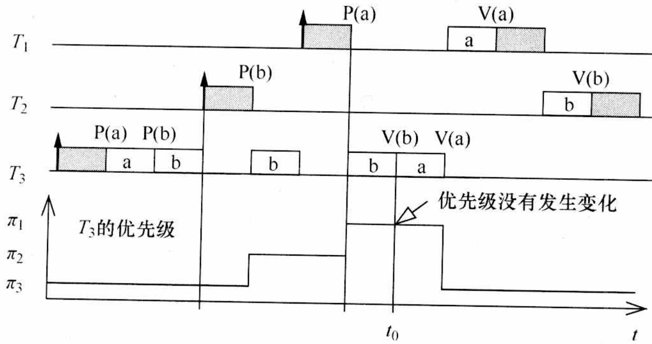



图 4.9 优先级嵌套

注意，在 $t_0$ 时刻，任务 $T_3$ 的优先级并没有恢复到它最初的优先级。反之，它的优先级提高到了它所阻塞的任务的最高优先级，在这一时刻，它的优先级为 $T_1$ 的优先级 $\pi_1$。

优先级继承的传递如图 4.10 所示 [Buttazzo，2002]。

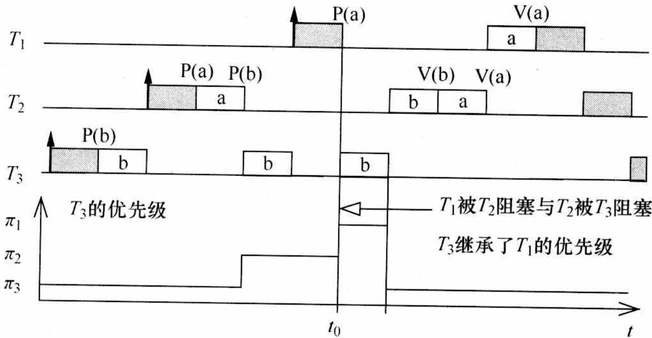



图 4.10 优先级继承的传递

在 $t_0$ 时刻，$T_1$ 被 $T_2$ 阻塞，$T_2$ 又被 $T_3$ 阻塞，因此 $T_3$ 继承了 $T_1$ 的优先级 $\pi_1$。

优先级继承也使用在 ADA 中，当许多任务同时访问一个临界资源时，每个任务的优先级都被设置为优先级的最大值。

优先级继承也解决了火星漫步者的问题：VxWorks 操作系统使用了一个标志位作为互斥的原语。这个标志位被打开的时候，就允许优先级的继承。但是当软件在火星车上开始运行时，这一位的初始状态被设置为了关闭。通过使用 VxWorks 的调试工具，将火星漫步者的这个标志位设置为打开，虽然此时火星漫步者已经在火星上了 [Jones，1997]。Levi 模拟软件也可以用来模拟优先级继承的情况 [Sirocic and Marwedel，2007c]。

虽然优先级继承能够解决很多实时系统问题，但是还不能解决所有的实时系统的问题。可能会有大量的任务拥有高优先级，甚至还有可能会产生死锁。**优先级冲**

顶协议[Sha et al. , 1990] 可能会对此有些帮助，但是这需要任务在设计时就对这些情况有一定的预知。

## 4.2 ERIKA

有些嵌入式系统（例如汽车系统以及智能家具系统）需要所有的任务都在微处理器的托管下运行[⊖]。出于这个原因，由这些系统的固件中的操作系统所能提供的服务必须是一个最小功能集，允许多线程执行周期性的和非周期性的任务，并且要对共享资源的竞争做好充分准备，以避免优先级反转的现象。

上述这些功能需求在1990年由OSEK/VDX [OSEK Group, 2010] 联合定义形成，它们定义了多线程实时操作系统的最小服务集，并且在8bit的微控制器上实现了一个体积为1~10KB的系统。OSEK/VDX的API最近被AUTOSAR [AUTOSAR, 2010] 组织进行了扩展，扩展中加入了对时间系统的保护、调度表时间触发系统以及用于保护由微处理器托管下的多任务之间内存保护的机制。本节简单地描述了这些系统的主要功能和需求，并基于一个已经实现了的开源实时操作系统ERIKA [Evidence, 2010] 作为参考。

OSEK内核区别于其他操作系统的第一个特点就是所有内核的部件在编译时静态定义。特别是，大部分该类系统都不支持动态的内存分配以及动态的线程创建。为了帮助用户去配置整个系统，OSEK/VDX标准提供了一个名叫OIL的配置语言，指定了必须在应用中实例化的对象。当应用被编译时，OIL编译器产生操作系统的数据结构，分配准确的、实际需要的内存的大小。这种方法的好处是应用程序需要多少内存就分配多少内存，并放在闪存中（在大多数的微处理器中，闪存的价格要比RAM更便宜）。

OSEK/VDX系统的第二个突出特点是堆栈的共享。造成堆栈共享的原因是RAM对于微处理器来说过于昂贵。如何实现堆栈共享的系统的关键在于如何编写任务代码。

对于传统的实时系统来说，一个周期性任务的典型实现一般根据如下的方案：

```
task(x) {
 int local;
 initialization();
 for (;;) {
  do_instance();
  end_instance();
}}
```

⊖ 这部分是G. Buttazzo和P. Gai（Pisa）提出的。

上面的方案具有大部分实时进程的特点：一个死循环，其中包含一个周期的任务，这个任务收到一个阻塞原语（end-instance（）），这个阻塞原语会阻塞任务，直到任务下一次被激活。当看到这样一种编程架构（在OSEK/VDX中叫做扩展进程）时，堆栈一直被当前这个进程占据，甚至进程休眠时也是这样。在这种情况下，堆栈是不能被共享的，这就意味着，每个任务都要有一个独立的堆栈。

当然，OSEK/VDX标准也对基本的任务提供支持，这种任务执行起来更像是一个函数的执行，其编程架构如下：

```
int local;
Task x() {
 do_instance();
}
System_initialization() {
 initialization();
 ...}
```

相比较于扩展任务，在基本任务中，当前状态必须由不同的实例进行维护，并且这些实例都不存放在栈中，而是存放在全局变量中。并且初始化的部分也放到了系统的初始化部分中，因为任务并不是动态创建的，所以这些初始化的信息可以在系统启动时就存在着。最后，任务的下一阶段运行并不需要通过同步原语的控制进行，因为任务实例化以后就已经一直在运行了。另外，任务不允许调用任何的阻塞原语，因此任务要么被更高优先级的任务抢占，要么就一直运行到任务结束。这样，任务就像一个函数一样，在栈上申请空间，运行，然后释放空间。出于这个原因，任务在两次执行之间，就不需要一直占据着堆栈，允许堆栈被系统所有的进程所共享。ERIKA支持堆栈共享，允许所有的基础任务在同一系统中共享一个堆栈，这样就可以减少RAM的大小。

OSEK/VDX内核在任务管理上提供了固定优先级调度系统和即时优先级上限的方法，以避免优先级反转的问题。使用即时优先级上限功能需要在OIL的配置文件中的每个资源的使用部分进行配置。OLI编译器通过每个任务的OIL文件的资源使用声明来计算整个系统的使用资源的上限。

另外OSEK/VDX支持非抢占式调度以及抢占阈值以限制整体堆栈的使用。限制任务之间抢占的主要原因是降低在同一时间申请系统堆栈的任务的数量，更进一步降低对整个RAM的需求。降低系统的抢占性有可能会降低整个任务的可调度性，因此系统的抢占程度必须衡量系统的调度性以及整个系统的可用RAM空间。

关于小型微控制器上的操作系统设计的另一个要求就是可扩展性，这意味着API的封装相对于标准系统来说，要尽可能精炼，以实现在小内存系统上的运行。实际上，在大规模生产的系统中，内存的总量影响到系统的总成本。在此背景下，

提出了一个一致性级别的概念用以对可扩展性进行衡量。该概念用于对操作系统的API特定子集进行定义。一致性级别可以随着系统的升级而进行扩展，对于最终的系统目标可以实现API封装的动态增减。OSEK/VDX标准所提供的一致性级别（由ERIKA公司）如下：

1）BCC1：这是一致分类的最小内核，提供了最少8个不同优先级的进程以及一个共享资源。

2）BCC2：相对于BCC1来说，这次的分类添加了可以拥有多个具有相同优先级任务的功能。每个任务都可以等待被激活，换言之，操作系统记录被激活，但尚未执行的任务的数量。

3）ECC1：与BCC1相比，这次的分类加入了可以等待某些特殊事件发生的扩展任务。

4）ECC2：这次的分类加入了多种激活以及扩展任务方式。

ERIKA公司通过提供下面两种一致性分类提供了更多的一致性类的种类：

1）EDF：这种一致性分类不使用固定优先级的调度器，而是使用为微处理器进行优化的、最早到期优先（EFD）调度器（见6.2.2.3节）。

2）FRSH：这种分类方式是EDF调度器的扩展，通过基于IRIS调度算法，提供一个资源预定调度器［Marzario et al.，2004］。

OSEK/VDX系统另一个有趣的特性是系统提供了一个API以用于中断的控制。这是OSEK/VDX与POSIX类系统最主要的不同之处，对于POSIX系统来说，中断是操作系统的禁区，并且没有任何操作系统的API可以用于中断控制。提供API控制中断的一个基本原因是微控制器的用户经常想要直接去控制中断的优先级，因此能够提供一种标准的方法去控制中断的使能就显得非常重要了。此外，OSEK/VDX标准指定了两种中断服务程序（Interrupt Service Routines，ISR）的类型：

1）分类1：更简单而且更快捷，在中断服务程序的结尾部分不去调用调度器。

2）分类2：中断服务程序可以调用一些原语改变调度器的行为。中断服务程序的结尾处是重新调度的调度点。中断服务程序1总是比中断服务程序2的优先级要高。

OSEK/VDX内核的另一个特点是可以移除产品化后系统中的一些操作检查点和错误检查码，除此之外，还可以定义钩子函数，当系统中发生指定的事件时，该函数被调用。当系统产品化后，移除这些检查点可以让系统变得效率更高（而且更安全）。

为了支持更好的调试体验，OSEK/VDX标准定义了一种命名为ORTI的文本语言，该语言描述了操作系统的各种对象是在哪里被分配的。ORTI文件一般由OIL编译器生成，并且由调试者使用，以打印一些定义在系统中的操作系统对象的详细信息（例如，调试者可以打印应用程序中的任务当前状态表）。

所有这些定义在OSEK/VDX中的标准，都被开源的ERIKA企业版内核［Evidence，2010］实现了。该内核支持一系列的嵌入式微控制器，并且目标代码的大

小是1~5KB不等。并且，ERIKA企业版实现了一些其他的功能，例如EDF调度器，提供了一个开放而且免费的操作系统可以用来学习、测试以及实现一些真实的用例，用于工业以及教育领域。

## 4.3 硬件抽象层

硬件抽象层（Hardware Abstraction Layer，HAL）提供了一种通过硬件无关的API访问硬件的方法。例如，可以提出一种硬件无关的技术，用来访问时钟而无需考虑时钟被映射到了哪块地址空间上。硬件抽象层主要用于硬件和操作系统层之间，它们提供了一种软件的知识产权（Intellectual Property，IP），但是它们既不是操作系统的一部分也不能归类于中间件。这方面的相关工作由Ecker、Müller和Dömer提供［Ecker et al.，2009］。

## 4.4 中间件

用于通信的库针对于缺乏通信机制的语言提供了额外的通信手段，它们在操作系统的基础之上提供了通信功能。由于添加在了操作系统之上，它们可以独立于操作系统（而且明显也独立于底层的处理器硬件）。结论就是，可以得到一个网络化的嵌入式系统。不仅仅局限于在系统内部通信，通信的距离越来越远也是一种未来的发展趋势。互联网协议的使用也变得越来越受欢迎。

### 4.4.1 OSEK/VDX COM

OSEK/VDX COM是用于OSEK车载操作系统上的通信标准［OSEK Group，2004］[1]。OSEK COM提供了“交互层”以用于API。无论是内部通信（ECU的内部通信）还是外部通信（与其他ECU通信），都可以使用这套接口标准。OSEK COM只指定了交互层的功能。符合规范的实现必然是分别发展的。

与其他ECU的交互层的通信要通过“网络层”以及“数据链路层”。OSEK COM指定了一些对这些层的要求及需求，但是这些层自己并不是OSEK COM的一部分。通过这种方式，就可以在不同的网络协议上实现通信。

OSEK COM是一个专用于嵌入式系统中间件通信的例子。除了专用于嵌入式系统的中间件外，许多用于非嵌入式系统的通信标准也同样适用于嵌入式系统。

### 4.4.2 CORBA

CORBA®（Common Object Request Broker Architecture，公共对象请求代理体

[1] OSEK是Continental Automotive公司的商标。

系结构）[Object Management Group (OMG), 2003] 可以视作一个采用上述标准的示例。CORBA 适用于远程的访问服务。通过 CORBA，可以使用标准的接口访问远程的对象。客户端使用本地的存根通信，模拟访问远端的对象。这些客户端发送要访问的对象的参数信息（假如有的话）至目标请求中介（Object Request Broker，ORB，见图 4.11）。ORB 接下来决定要访问对象的所在地并且通过标准协议来发送信息（例如 IIOP）至对象所在地。这些信息通过具体的链路被发送至对象中，当对象收到信息后，也会通过 ORB 将对应的请求信息返回。

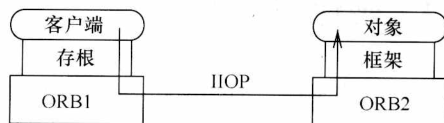



图 4.11　使用 CORBA 访问远程对象

标准的 CORBA 不为实时应用提供可预测性，因此，实时的 CORBA（RT-CORBA）标准在此情况下被提出 [Object Management Group (OMG), 2005a]。RT-CORBA 的一个基本特点是在固定优先级的系统中提供点到点的时间可预测性。它包括了考虑客户端和服务器端争夺资源的线程优先级，并且设定一个操作调用的延迟的边界。

实时系统的另一个典型问题是当线程发生了对临界资源的争用时，线程的优先级此时有可能失去作用。RT-CORBA 必须解决优先级反转的问题。RT-CORBA 包含了一种当优先级反转发生时，提供一个时间边界的方法。RT-CORBA 提供了管理线程优先级的功能，虽然这个功能可以与 POSIX 标准的实时扩展功能相容，但这个优先级独立于操作系统底层的优先级。客户端线程的优先级可以被传送到服务器端，优先级管理也可以用于管理临界资源的原语。刚刚描述过的优先级继承的协议是已经在 RT-CORBA 中实现并且可用的。线程池的存在避免了线程创建时的开销。

### 4.4.3 MPI

作为 CORBA 的一个替代方案，信息传送接口（Message Passing Interface，MPI）也可以用来在不同的处理器之间传递信息。MPI 作为一个使用非常频繁的库，最初是被设计用于高性能计算。MPI 基于消息传递的，并且允许在同步和异步消息传递之间进行选择。例如，使用 MPI 库的发送命令发送同步消息 [MHPCC, 2010]：

MPI_Send（buffer，count，type，dest，tag，comm），其中：

1）buffer：要发送数据的地址。

2）count：要发送的数据数量。

3）type：要发送的数据类型（例如，MPI 字符、MPI 短整型、MPI 整形）。

4）dest：要发送的目标进程的进程 ID。

5）tag：消息的 ID（用来对传入消息进行排序）。

6）comm：通信的上下文（一组处理过程，用来判定消息目的地是否合法）。

7）Function result：成功与否的返回值。

下面是异步消息传输的库函数：

MPI_send（buffer，count，type，dest，tag，comm，request），其中：

1）buffer，count，type，dest，tag，comm：同上所述。

2）这里的系统都有一个独特的“request number”。程序员使用这个系统分配“handle”（在WAIT类型中）用以确定非阻塞的操作是否完成。

对于MPI而言，在不同的处理器之间分区计算必须要明确的完成，并且通信和数据的分布也同样如此。同步过程一般都隐含在了通信中，但是明确的同步也是有可能发生的，结果程序员要把大量的工作放在对代码的管理中。因此也不能对众多发生变化的进程进行衡量［Verachtert，2008］。

为了在实时系统中部署MPI方式的通信，一个被作MPI/RT的MPI实时版本被定义［MPI/RT forum，2001］。MPI-RT没有覆盖RT-CORBA的所有方面，例如线程的创建和终结。MPI/RT被认为是在操作系统和标准（非实时）MPI之间的一个潜在的层。

MPI可以在不同的平台上使用，并且也可以在片上多处理器系统中运行。然而，它基于了内存访问的速度要比通信操作的速度要快得多的假设。另外，MPI主要针对同质多处理器，这些假设并不适用于片上多处理器系统。

### 4.4.4 POSIX线程（Pthreads）

POSIX线程库是操作系统级的线程应用编程接口（API）［Barney，2010］。Pthread遵守IEEE POSIX 1003.1c操作系统标准，一组线程可以运行在同样的地址空间中，因此就可以基于共享内存实现通信了。这样就避免了MPI中需要做到的内存复制操作，因此这个库就更适用于共享同一地址空间的多核处理器编程。POSIX库也包含了关于互斥操作的标准API。Pthread使用完全明确的同步［Verachtert，2008］，具体的语义取决于内存一致性模型的使用。正确地进行同步编程是一件比较困难的事情，POSIX库可以作为其他编程模型的后端。

### 4.4.5 OpenMP

对于OpenMP，并行性是其主要特征，计算划分、通信以及同步等，相对而言都是次要的。并行性需要添加一些编译指示来表达：例如，循环之前可以加入编译指示，以表明它们可以并行执行。下面的这段程序展示了一小段并行循环编程［OpenMP Architecture Review Board，2008］：

```
void a1(int n, float *a, float *b)
 {int i;
 # pragma omp parallel for
  for (i=1; i<n; i++) /* i 私有默认 */
   b[i] = (a[i] + a[i-1]) / 2.0;
 }
```

这意味着（在上面刚刚提出的方法）OpenMP需要一点点的工作就可以为用户提供并行化的编程，然而这也意味着用户不能很好地控制这些部分［Verachtert，2008］。OpenMG针对的是共享内存的硬件，这是MPSoCs上的第一个应用（可以参考［Marongiu and Benini，2009］的例子）。

### 4.4.6　UPnP、DPWS和JXTA

通用即插即用（UPnP）已经将即插即用的概念从PC发展到了设备与网络的互连中。很容易看到，通过网络互连家中和办公室的打印机、存储空间等已经是一个未来发展的基本趋势了［UPnP Forum，2010］。出于安全问题，只有数据之间的交换，代码是不能被传输的。

Web服务设备配置文件（Devices Profile for Web Services，DPWS）的目标是比UPnP做得更加通用。“DPWS被设计用来在资源受限的设备上使能安全的Web服务（WS）能力”［ws4d，2010］。DPWS指定了发现设备并连接到网络上的服务，用于交换可用的服务的信息以及发布、描述事件。

除此之外，几个综合的网络通信库可以用于高性能计算服务设计（High-Performance Computing，HPC）。这些库特别适用于低耦合的基于互联网的通信协议。JXTAT M［JXTA Community，2010］就是一个开源的、点对点的协议规范。这个协议由一组XML消息定义，它允许任何设备链接到网络上的某个端点进行信息的交换，并且可以与独立的网络拓扑结构进行合作。JXTA创建了一个虚拟的覆盖网络，允许一个点与其他的点进行交互，甚至当某些节点或者资源位于防火墙后，也可以进行交互。JXTA的名字源于单词“juxtapose”。

CORBA、MPI、Pthreads、OpenMP、UPnP、DPWS以及JXTA都是一些通信中间件（用于操作系统以及应用程序之间的软件层）。最初，这些中间件都被设计用来进行桌面电脑之间的通信，然而它们正在试图去影响嵌入式系统上的知识以及技术。对于一些移动设备，例如智能手机，使用上面的方法可能就是比较恰当的。对于“硬实时系统”来说，它们的开销、实时性以及所提供的服务就未必适合了。

## 4.5　实时数据库

数据库提供了一种方便、结构化的方法存储和访问信息。因此，数据库提供了API用于进行数据的读写操作。一系列的读和写的操作被称为事物，事物失败的原因有很多种：有可能是硬件原因所导致，有可能是死锁，或者是并发控制问题等。除非事物执行到了非常安全的结束时期，否则事物不会对数据库的状态产生任何影响。因此，由事物引发的变化通常不认为是一次处理的终结，除非事物已经被提交。大多数的事物一般都需要原子操作。这意味着，由事物产生的一些最终结果

（数据库的新状态）必须是要么就是事物完全执行完毕，要么就是事物跟未执行前的状态一致。另外，由事件引起的数据库状态必须是一致的。一致性需求包括，例如，相同的事物发出的读请求的值，必须是一致的（不要描述一个数据库模拟出来的不存在的环境）。此外，对于其他的数据库用户来说，没有中间状态造成的部分事物的执行必须是可见的（事物必须就像独立执行的那样被执行）。最终，事物对数据库的改变应该是永久的，这个属性又被称为事物的持久性。同时，上面用黑体标出来的四个属性被称作数据库的 ACID 属性（参见 Krishna 与 Shin [Krishna and Shin，1997] 所著书籍第 5 章）。

对于某些数据库来说，有软实时的限制，例如航空订票系统的时间限制就是软实时的。与此相反，也有一些硬实时限制的数据库，例如汽车应用中的自动行人识别系统以及军事上的目标识别系统软件，就必须是硬实时系统限制。上面所说的那些要求很难保证硬实时的限制。例如，在事物被提交前，事物就有可能在不同的时间被终止了。所有的数据库都依赖于页请求以及硬盘，访问硬盘的时间非常难以预测。一个可能的解决方法就是使用贮存数据库以及通过使用闪存进行预测。嵌入式数据库有时足够小，以确保能够使用上述方法。想了解更多的信息，请参考 Krishna 以及 Shin 所编写的书籍。

## 4.6　思考题

1. 都有哪些需求必须需要一个实时操作系统？如何区分标准操作系统与实时操作系统的需求的不同？

2. 自从 1958 年以来，一共有多少秒在除夕用来补偿 UTC 与 TAI 时间的不同？你可以通过互联网来寻找问题的答案。

3. 在 RTOS 中，类似于 Windows 或者 Linux 这样的标准操作系统的哪些特征会被抛弃？

4. 找出使用了内存保护单元的处理机！内存保护单元与更加常用的内存管理单元（MMU）的区别是什么？你可以通过使用互联网来搜寻相关的答案。

5. 描述一下这么多种类的嵌入式系统，哪种保护方式必须被提供！描述一下这些系统中，哪种保护方式我们可以不需要提供！

6. 举一个具有三个任务的系统中，产生优先级反转的例子。

7. 从 Levi 的网站 [Sirocic and Marwedel，2007c] 所下载的 Levi 的学习模块 LeviRTS 模拟如图 4.12 描述的进程集。

| 任务 | 优先级 | 目的地 | 运行时间 | 打印 | | 指令序列 | |
|---|---|---|---|---|---|---|---|
| | | | | $t_{P,P}$ | $t_{V,P}$ | $t_{P,C}$ | $t_{V,C}$ |
| $T_1$ | 1(高) | 3 | 4 | 1 | 4 | - | - |
| $T_2$ | 2 | 10 | 3 | - | | 1 | 2 |
| $T_3$ | 3 | 5 | 6 | - | - | 4 | 6 |
| $T_4$ | 4(低) | 0 | 7 | 2 | 5 | - | - |

图 4.12　任务专用资源请求设置

$t_{P,P}$以及$t_{P,C}$是相对于开始的时间，这里有任务分别独立请求使用打印系统或是通信线路（在 Levi 中被称作 $\Delta\ tP$），$t_{V,P}$以及$t_{V,C}$与资源被释放的时间有关。使用基于优先级的抢占调度将会导致什么问题发生？如何解决这个问题？

8. 在网络中间件的设计中，发生优先级反转会产生什么样的影响？

9. 闪存会对实时数据库的设计产生什么样的影响？

# 第 5 章　评估和验证

## 5.1　简介

### 5.1.1　范围

规格说明书、硬件平台以及系统软件告诉人们，究竟可以用哪些基础材料来设计嵌入式系统。在设计过程中，必须相当频繁地去对设计进行评估以及验证。因此，在进入设计阶段之前，必须描述如何进行评估以及验证。尽管评估和验证之间有很多区别，但是它们有着相当紧密的联系。

**定义：验证**是检查一个特定的（某些部分）设计是否符合需求、符合所有的约束以及按照预期执行的一个过程。

**定义：**经过数学严格计算过的验证被称作核实。

对于任何程序设计来说，验证都是非常重要的。几乎没有任何系统能够在设计阶段不经过验证就能够按照预期运行。在安全相关的嵌入式系统中，验证更显得极其重要。理论上，可以尝试去设计一个总是可以根据规格说明书而形成正确实现的验证工具。实际上，除了一些极其简单的场景外，这种验证工具几乎无法满足要求。因此，每一个设计都要进行验证。为了最小化必须验证的设计的数量，可以尝试在设计过程的最后去进行验证。不幸的是，这种方法也经常不能工作，由于在基于规格说明书的、抽象的系统设计阶段与实现阶段有着巨大的差别，因此就需要在不同的设计阶段进行验证工作（见图 5.1）。验证和设计工作应当交织在一起进行，而不能认为是两个相互独立的活动。

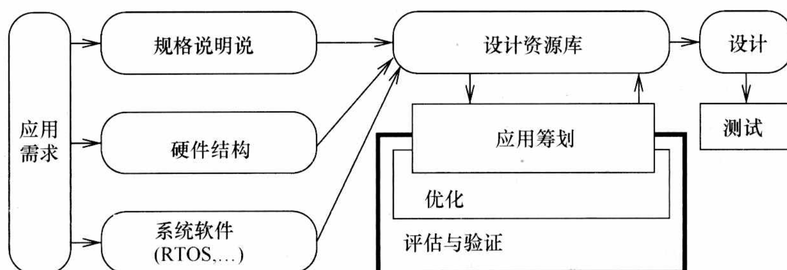



图 5.1　本章所处的上下文环境

假如能够有一个能够适用于所有验证问题的验证工具就好了。实际上，没有哪个可用的技术能够解决所有的问题，一般都是若干种技术混合使用。从本章的5.6节开始，会展示一系列可使用的关键技术，这些技术会让人们对评估技术有一个总体的了解。

**定义**：评估是对一个特定（有可能是部分）设计的一些关键特征（或者是目标）进行信息的定量计算的过程。

## 5.1.2　多目标优化

设计评估通常会导致对同一个特征产生出若干的设计方案，例如平均执行时间和最差执行时间的案例、系统的功耗、代码的大小、可靠性以及安全性。将所有这些条件放入一个目标方案（例如通过使用加权平均数）中通常是不现实的，因为这将隐藏某些设计的基础特征。当然，明智之举是让设计师来决定，究竟哪组设计方案更加可行。这些设计中应该只包含合理的设计。**多目标优化技术**的目的就是找到这样一组合适的设计。

为了执行多目标最优化设计，认为一个 $m$ 维度的空间 $X$ 是优化问题的可行解决方案。这些维度可以反映一些如处理器的数量、内存的大小、总线的类型等。对于 $X$ 空间来说，定义了一个 $n$ 维的函数，评估就若干标准及目标进行设计（例如系统开销以及性能）：

$$f(x) = (f_1(x), \cdots, f_n(x))$$

式中，$x \in X$。

$F$ 是这些目标在 $n$ 维空间的值（也被称作目标空间）。假设，对每个目标来说，最终的总的请求 < 对应的反馈 ≤ - 被定义的请求。接下来，假定目标是最小化的对象。

**定义**：向量 $\boldsymbol{u} = (u_1, \cdots, u_n) \in F$ 支配向量 $\boldsymbol{v} = (v_1, \cdots, v_n) \in F$，如果 $\boldsymbol{u}$ 比 $\boldsymbol{v}$ “适用”：

$$\forall i \in \{1, \cdots, n\}: \ u_i \leqslant v_i \wedge \tag{5.1}$$

$$\exists i \in \{1, \cdots, n\}: \ u_i < v_i \tag{5.2}$$

**定义**：如果 $\boldsymbol{u}$ 与 $\boldsymbol{v}$ 都不属于 $\boldsymbol{v}$，则向量 $\boldsymbol{u} \in F$ 被称作与向量 $\boldsymbol{v} \in F$ 是不相关的。

**定义**：如果没有 $y \in X$ 导致 $\boldsymbol{u} = f(x)$ 属于 $\boldsymbol{v} = f(y)$，则 $x \in X$ 被称作与 $X$ 有关的帕累托最优（Pareto-optimal）。

之前的公式定义了在解空间的帕累托最优。接下来的公式可以用于目标空间的系统优化。

**定义**：让 $\boldsymbol{S} \subseteq F$ 作为目标空间的一组向量。如果 $\boldsymbol{v}$ 不是属于 $S$ 的任何元素，则 $\boldsymbol{v} \in F$ 被称作与 $\boldsymbol{S}$ 相关的非控解。如果 $\boldsymbol{v}$ 不属于解决方案 $F$ 的任意自己，则 $\boldsymbol{v}$ 被称作帕累托最优。

图5.2给出了相对于解空间，突出了目标空间的不同方面。

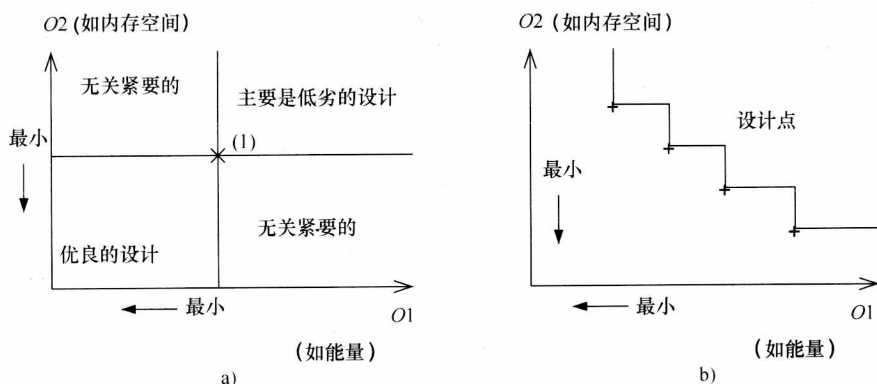



图 5.2　帕累托点和帕累托前沿

a）帕累托点　b）帕累托前沿

右上方的区域与设计空间的优化相对应（1），因此处于此空间的方案与其他方案相比效率较差。左下方矩形空间（如果此处有元素存在）的效率要优于其他的空间坐标域。位于左上角和右下角的元素无需关注，这些元素仅仅用于与较差的或者较好的优化方案进行对比使用。图 5.2b 显示了一组帕累托点，也就是所谓的帕累托前沿。

基于帕累托点的**设计空间研究**(Design Space Exploration，DSE）可以帮助设计者找到帕累托最优的设计方案，也允许设计者在大量的可用的实施方案中进行选择。

## 5.1.3　相关目标

对于 PC 类系统来说，在新系统的设计中，平均性能预测扮演着主导角色。对于嵌入式物理子系统来说，需要更多地考虑多目标处理。下面的各向说明了这些目标是否以及在本书的哪里进行讨论：

1）**平均性能**：对目标的分析经常基于仿真。5.6 节将简单地提出一些有关仿真的问题。大量的在仿真系统上的（特别是在异构系统、物理子系统）有关信息是可以见到的。由于大量的物理效应，想提出一个完整的参考列表几乎是不可能的。

2）**最坏情况下的性能/实时行为**：在 5.2.2 节，将会看到 aiT 时间分析工具的详细介绍。

3）**功耗**：关于这一技术的简要概述将在 5.3 节予以呈现。

4）**温度/热行为**：将在 5.4 节中就此主题做一个简单的介绍。

5）**可靠性**：在 5.5 节中，可以找到有关可靠性理论的介绍。

6）**电磁兼容性**：本书将不会提及此类内容。

7）**数值精度**：在一些应用程序中，一些小的数值精度误差是可以容忍的。容

忍这种精度上的损失可以提升其他部分的设计。例如，将会在7.2.1节讨论浮点数到定点数结构的转换。还有一些其他的类似情况也是存在的。

8）**可测试性**：测试系统的成本可能非常巨大，有时甚至会超过生产成本。因此在进行系统设计时就要开始考虑测试问题。将会在第8章中讨论测试。

9）**成本**：本书不考虑硅面积以及实际的金钱花费。

10）**开销、鲁棒性、易用性、扩展性、保密性、安全性、友好性**：本书也未提及此类相关内容。

当然，实际的需求有可能比上面列出的内容要更多。接下来的内容将提出一些基于最差情况的性能评估方法。

## 5.2 性能评估

性能评估旨在预测系统的性能。这是一个重大的挑战（特别是对于物理子系统而言），因为可能需要对最坏的情况进行预测，而不仅仅是通常系统的整体性能。这些信息是必需的，因为必须要保证系统的实时约束。

### 5.2.1 早期阶段

有两种技术方法可以用来为了在系统的早期设计阶段就获得性能信息：

1）**估算成本和性能值**：相当数量的评估方法已经基于此目的被开发，例如Jha与Dutt［Jha and Dutt，1993］为硬件所作的此类工作，Jain等人［Jain et al.，2001］以及Franke［Franke，2008］在软件上所作的此类工作。若想估算的值尽量准确，需要相当大的努力才行。

2）**实际成本和性能值**：可以使用真正的软件代码（以二进制的形式）在非常接近真实硬件的平台上运行。假如“软件合成工具”（编译器）和硬件合成工具的接口存在，这几乎是惟一的可能。这种方法相对于之前的方法估算更加的精确，但会较大地（有时是极大地）增加时间开销。

为了获得足够准确的信息，通信开销也应该被考虑进来。但不幸的是，很难在系统的早期设计阶段去计算通信的消耗。

### 5.2.2 WCET估算

研究人员提出了一种正式进行性能评估的技术。在嵌入式系统上，Thiele等人、Henia和Ernst等人以及Wilhelm等人做了与此相关的研究（参见［Thiele 2006b］，［Henia et al.，2005］，and［Wilhelm，2006］）。这些技术需要对一些知识架构有了解，这些技术并不适合在系统的极早期阶段介入，但是其中的一些方法可以在对目标系统没有详细了解时就使用。这些方法模拟真实、物理的时间。

任务调度需要对任务的执行时间有一定的了解，特别是在实时系统中，要保证

时间约束性。**最坏情况下执行时间**（Worst Case Execution Time，WCET）是大多数调度算法的基础。WCET 的相关定义如图 5.3 所示。

最坏情况下执行时间是一个程序从有任何输入和最初的执行状态下最长的执行时间。但不幸的是，WCET 一般都极其难以进行计算。一般来说，很难判定 WCET 是否是有穷的。显而易见的事实是，它无法判定一个程序是否结束，因此 WCET 只能用于某些程序/任务的计算。WCET 只能计算如没有递归的程序、没有 While 循环的程序以及有明确的迭代数的循环。假如有上述的限制，计算 WCET 几乎是一件不可能完成的任务。现代处理器的流水线架构带来的危害以及存储的结构导致有限的可预计的命中率使得几乎无法在设计时进行准确的预测。计算系统的 WCET 包含缓存、流水线、中断以及虚拟内存的预测，这是一个更大的挑战。所以，如果能够计算出 WCET 的上界，那么这真是一件值得高兴的事情。

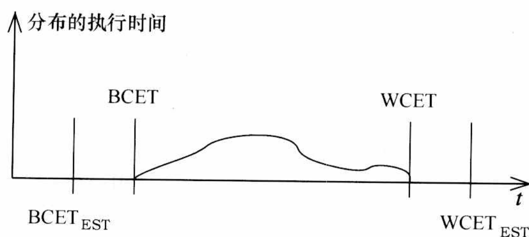


图 5.3　WCET 相关条款

这个上界通常称为**估算的最坏情况执行时间**，或者 $WCET_{EST}$。这种边界应该具有至少两种属性：

1）边界应该是安全的（$WCET_{EST} \geq WCET$）；

2）边界应该严格（$WCET_{EST} - WCET \leq WCET$）。

注意，术语“估计”并不意味着得到的时间是不安全的。

有时，架构的特征也会降低平均执行时间，但是这并不能保证在实时设计中完全忽略 WCET。计算精准的执行上界的时间依旧很困难。上面提到的设计架构在计算 $WCET_{EST}$时依旧存在困难。

因此，**最佳执行时间**（Best-Case Execution Time，BCET）以及相对应的估算 $BCET_{EST}$以类似的方式被定义。$BCET_{EST}$是一种安全并且严格的执行时间的下界。

通过使用高级语言，例如 C 语言而没有任何汇编语言的知识或者是底层平台架构的知识而去计算紧界是不可能的。因此，安全的分析必须从真实的机器码开始，任何其他的方法都会导致不安全的结果。

接下来将要更加紧密地学习 WCET 估算。接下来将描述了基于 R. Wilhelm [Wilhelm，2006] 所开发的工具 aiT，aiT 的结构如图 5.4 所示。

与所讨论的高级语言的问题一致，aiT 由代码组成的可执行文件开始进行分析。从这些代码中可以提取出控制流图（Control-Flow Graph，CFG）。接下来循环转换被应用，它包括对循环和递归函数调用的转换以及虚拟循环展开。被称作“虚拟”展开的原因是它是在内部执行的，而没有实际修改过的代码运行。它可以用 CRL（Control flow Representation Language，控制流表示语言）格式代表。接下

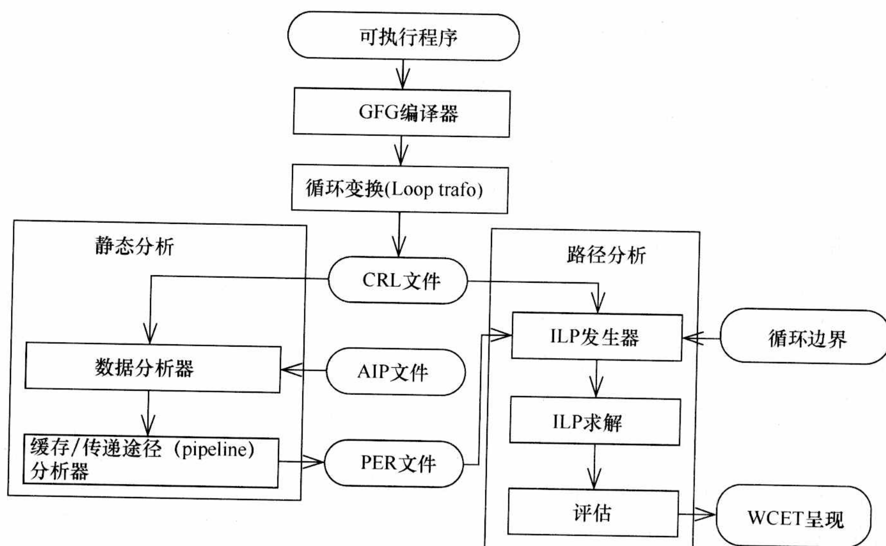



图 5.4　aiT 时间分析工具架构

来的阶段采用了不同的静态分析，静态分析读取包含设计者注释的 AIP 文件。这些注释所包含的信息很难或者说无法自动的从代码中提取（如复杂的循环边界）。静态分析包括了数值、缓存以及流水线的分析。

**数据分析**计算一个封闭间隔内的寄存器以及本地变量中可能出现的值。其结论可用于控制流分析以及数据缓存分析。通常来说，像地址这样的信息一般都非常精确地知道了（特别是对“干净的”代码来说），这对内存访问的预测非常有帮助。

下面是对缓存以及**流水线**的分析。在下面将介绍一些有关缓存分析的细节。

假设使用 $n$ 路的组相关缓存（见图 5.5）[1]。

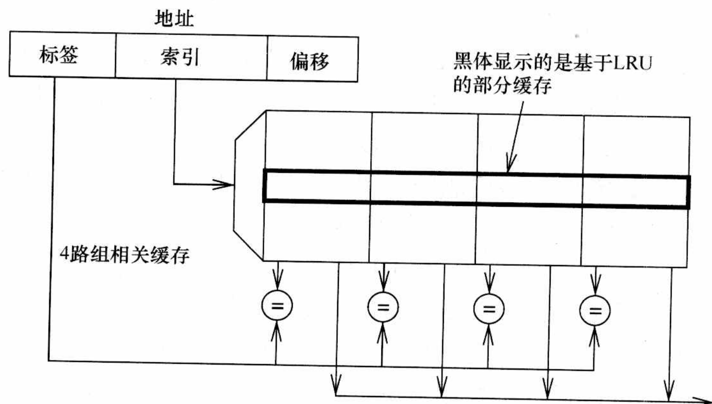



图 5.5　组相关缓存（$n=4$）

[1] 假设本书的读者对于缓存的相关概念非常熟悉。

这里认为这部分（行）的缓存与特定的索引相对应（见图 5.5 黑体部分）。假定被换出缓存的那部分数据是被近期最少使用策略（Least Recently Used，LRU）换出的。这意味着所有的缓存中的数据都分别对应着一个它们各自的、特殊的引用。最后的 $n$ 意味着内存块就存放在缓存的这些部分中。这里认为 LRU 硬件管理着所有的索引，并且每个索引都与其他索引都是独立的。在这种假定下，每一块特定的索引的换出都是与其他索引的状况完全独立的。这种独立性是非常重要的，因为它允许人们独立地去考虑每一个索引。

现在考虑一部分缓存以及一个特定的索引。假如有每一个缓存通路（列）的信息。此外，考虑到控制流的加入。当部分缓存加入后，能知道什么呢？必须在可能以及必需的信息中进行相应的分析并分辨。必须的分析显示了必须在缓存中的条目。在计算 WCET 时，这些信息就非常有意义了。可能的结果表明了有可能在缓存中的条目。这些信息特别用于推断某些特定的信息确实不在缓存中。这些知识在计算 BCET 中特别有效。作为“必须”以及“可能”的分析的例子，认为“必须”的信息位于控制流中。图 5.6 显示了相应的情况。左边的这些条目被看作比右边的条目后进入缓存。

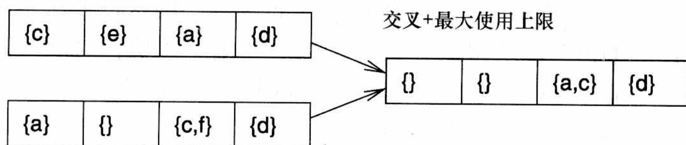



图 5.6 LRU 缓存在程序结合点的必须分析

在图 5.6 中，内存对象 c 被看作某一路结合点上最新的对象，此外 a 被看作另一路结合点上最新的对象。其他条目的使用时间也类似定义。知道在加入以后的“最坏”情况是什么吗？只有在确保两个通路的条目都存在于缓存中时，才能保证该条目存在于缓存中。这意味着相交的内存对象决定了加入“必须分析”的结果。作为最坏情况的例子，必须让两条通路都承担着最长时间的数据存在。图 5.6 显示了结果，很明显，该分析必须基于每组缓存通路的条目集合。

接下来继续讨论可能的分析控制流接口。图 5.7 描述了这一场景。

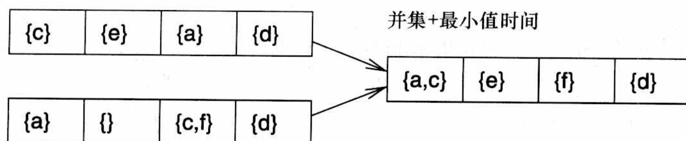



图 5.7 LRU 缓存在程序结合点的可能分析

在两条通路的某一条加入通路后，缓存中的数据有可能被加入。这意味着对象在进入缓存后，一组对象的联合也会被加入到缓存中。在最佳情况下，可以在加入

后使用最少时间。图5.7显示了这个结果。

对于内存块b的引用来说，被访问的内存块移动至时间最新的位置上，此外其他的内存块的进入时间加1。

静态分析还包括了对流水线的分析。流水线分析必须计算在机器流水线上执行的机器码循环次数的边界。R. Wilhelm [Wilhelm，2006] 以及 S. Thesing [Thesing，2004] 对流水线分析做了细致的陈述。

静态分析的总体结果由每个程序的基本块执行时间的边界组成。在图5.4所示的PER文件图中写入了结论。

aiT的下一个阶段使用这些边界是为了获取整个程序的最坏执行时间，这些步骤基于ILP模型。在这些模型中，总的运行时间用于目标函数中。总的执行时间的计算通过对基础模块的预期执行时间乘以它们的执行频率得到。基本块的执行时间定义为这个块的单次执行WCET（作为静态分析进行计算）乘以最坏情况下执行的块的数量。只有部分执行块的数目可以被自动确定。因此，确定ILP模型依赖于额外的设计者提供的信息，例如循环边界。这些信息可以从外部的AIP文件中获得。约束块之间的关系模型，这种对执行时间进行模拟的技术被称作**隐式路径枚举**，这种方法避免了对大量潜在的可执行路径的执行。用这种方法定义的ILP问题可以通过一些标准的ILP解决最大化目标函数。使用总体执行时间可以生成最大的生产效率的安全上界。aiT还可以通过带注释的控制流图提供一个可视化的结果。设计者可以分析这些图标用来优化设计系统。

### 5.2.3 实时微积分学

Thiele的实时微积分(Real-Time Calculus，RTC) 以对传入事件的速率的描述作为依据[⊖]。这些描述中也包含着对波动率的描述。为达到这样的目的，时间序列事件的特性（或流）可以被表示为一个元组到达的曲线：

$$\overline{\alpha}^{u}(\Delta),\overline{\alpha}^{l}(\Delta)\in \mathrm{IR}\geqslant 0,\Delta\in \mathrm{IR}\geqslant 0$$

这些曲线代表了最大位移。最小数量事件到达的时间间隔为$\Delta$。对于所有的$t\geqslant 0$来说，在某一时间间隔中（$t$，$t+\Delta$），最多有$\overline{\alpha}^{u}$（$\Delta$）个事件发生，最少有$\overline{\alpha}^{l}$（$\Delta$）个事件发生。图5.8显示了基于某些可能到达事件的模型可能到达事件的数量。

例如，在以$p$作为周期的周期事件流到来的情况下，在此时间间隔内，最多只能有一个事件发生（0，$p$）[⊜]。同样，两个时间的时间间隔也有一个上界（$p$，$2p$）。现在，假设时间间隔的下界（0，$p$），在此时间间隔内有可能一次事件都没有发

⊖ 所有有关实时微积分学的描述都是基于Zurawski所著书籍中Thiele在相关章节所作的描述 [Thiele，2006b]，在系统级所要考虑的因素被称为模块化的性能分析（Modular Performance Analysis，MPA）。

⊜ 不讨论$\Delta=n*p$不连续时的情况。

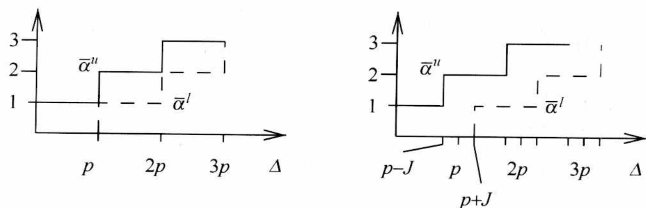



图5.8 到达曲线：周期性的事件流（左）和有抖动 $J$ 的周期性的事件流（右）

生，因此边界为0。对于时间间隔（$p$，$2p$），必须至少有一个事件，因此边界为1。所以，对于 $\Delta = 0.5p$，最少0个最多1个事件到来（见图5.8左图）。至于伴随着抖动 $J$ 的周期性事件流，曲线的周期随着此量移动。上界向左移动，下界向右移动。假定抖动不会积累，通过在符号上加入一个上标（例如 $\bar{\alpha}$）表示所有的事件。

可用的计算以及通信服务的功能可以由一组服务函数来描述：

$$\beta^u(\Delta), \beta^l(\Delta) \in \mathrm{IR} \geqslant 0, \Delta \in \mathrm{IR} \geqslant 0$$

这些函数允许人们去模拟可用的服务功能的波动情况。图5.9显示了一些时分多址（Time Division Multiple Access，TDMA）总线的通信功能。这种功能会周期性地分配一段时间 $p$。总线仲裁机制会将时间划分成的时间片分配给总线。在这段时间片中，总线会达到 $b$ 的带宽。

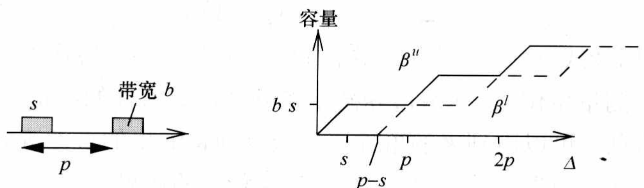



图5.9 TDMA总线服务功能

如果总线在进行观测时即被分配，则上界就可以被准确获得，转移的数量将会呈现线性的增长。如果在长度 $\Delta$ 期间进行观察时总线被释放，那么就可以获得对应的下界。此后必须在 $p$-$s$ 的时间内进行等待，直到总线被再次分配。

需要有单独的方法能够对 $\bar{\alpha}$ 和 $\beta$ 之间到达的事件流（外部）进行模型的建立。这种计算并不是RTC的一部分。与之相反，边界事件中形成的系统是由微积分推导得出的（见下面所述）。

到此为止，还没有每个到达事件的工作负载所需的信息。传入事件序列 $e$ 的工作负载由函数 $\gamma^u(e)$，$\gamma^l(e) \in \mathrm{IR} \geqslant 0$ 所表达。这个信息可由每个事件所需的代码执行时间的边界导出。图5.10显示了这些函数的相关示例。

该示例基于单个事件的处理时间在3～4个时间单元内，于是单个事件的工作负载就在3～4个时间单元内变化，两个事件的工作负载在6～8个时间单元内变化，等等。

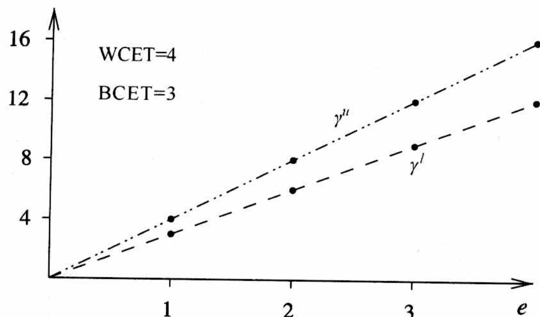



图 5.10 工作负载特征描述

虚线所表示部分与函数无关，它只定义了一个由整数代表的事件数量。由此，一个到来事件流的工作负载量就可以很容易地被计算出来。上界和下届的特征函数如下：

$$\alpha^u(\Delta)=\gamma^u(\overline{\alpha}^u(\Delta)) \tag{5.3}$$

$$\alpha^l(\Delta)=\gamma^l(\overline{\alpha}^l(\Delta)) \tag{5.4}$$

应该有足够的计算或通信能力来处理这些工作负载。有足够的计算能力被处理的时间可以计算为

$$\overline{\beta}^u(\Delta)=(\gamma^l)^{-1}(\beta^u(\Delta)) \tag{5.5}$$

$$\overline{\beta}^l(\Delta)=(\gamma^u)^{-1}(\beta^l(\Delta)) \tag{5.6}$$

式（5.5）以及式（5.6）使用了逆函数 $\gamma^u$ 以及 $\gamma^l$ 用于转换可用计算或通信能力的边界（实时测量单位），并由边界衡量能够处理的事件的数量。

基于这些信息，可以由到来的事件流中获得即将流出的事件流的属性。假设到来的事件流的特征是边界 $[\overline{\alpha}^l,\overline{\alpha}^u]$，可以依照相应的边界 $[\overline{\alpha}^{l'},\overline{\alpha}^{u'}]$ 计算流出的事件流的特征以及剩余的服务能力，以供其他任务使用。剩余的服务能力可以由服务曲线 $[\overline{\beta}^l,\overline{\beta}^u]$ 至服务曲线 $[\overline{\beta}^{l'},\overline{\beta}^{u'}]$（见图5.11）得出。这些剩余的服务能力可以用于统一处理器上的较低优先级的任务。

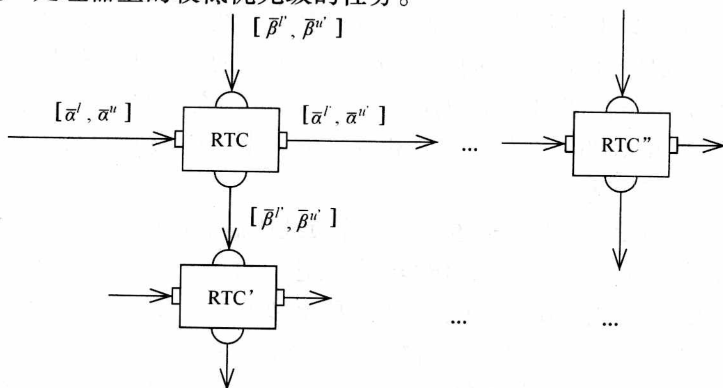



图 5.11 可用于实时组件的事件流的转换以及服务能力

Thiele 等人提到，流出的事件流以及剩余的服务能力受如下函数约束［Thiele, 2006b］:

$$\overline{\alpha}^{u'} = [(\overline{\alpha}^{u} \underline{\otimes} \overline{\beta}^{u}) \overline{\oslash} \overline{\beta}^{l}] \wedge \overline{\beta}^{u} \tag{5.7}$$

$$\overline{\alpha}^{l'} = [(\overline{\alpha}^{l}) \overline{\oslash} \overline{\beta}^{u}) \underline{\otimes} \overline{\beta}^{l}] \wedge \overline{\beta}^{l} \tag{5.8}$$

$$\overline{\beta}^{u'} = (\overline{\beta}^{u} - \overline{\alpha}^{l}) \underline{\oslash} 0 \tag{5.9}$$

$$\overline{\beta}^{l'} = (\overline{\beta}^{l} - \overline{\alpha}^{u}) \overline{\otimes} 0 \tag{5.10}$$

式中　$\wedge$——最小运算符。

用于这些方程式的操作符定义如下：

$$(f \underline{\otimes} g)(t) = \inf_{0 \leqslant u \leqslant t} \{f(t-u) + g(u)\} \tag{5.11}$$

$$(f \overline{\otimes} g)(t) = \sup_{0 \leqslant u \leqslant t} \{f(t-u) + g(u)\} \tag{5.12}$$

$$(f \overline{\oslash} g)(t) = \sup_{u \geqslant 0} \{f(t+u) - g(u)\} \tag{5.13}$$

$$(f \underline{\oslash} g)(t) = \inf_{u \geqslant 0} \{f(t+u) - g(u)\} \tag{5.14}$$

本质上，这些等式表达了流出的事件流以及系统服务能力。这些方程已被通信理论所采用，并由网络微积分所证明［Le Boudec and Thiran, 2001］。可以通过下载 Matlab 的工具箱，轻松使用这些方程［Wandeler and Thiele, 2006］。

同样的理论同样可以计算出实时组件所造成的延时，同时还可以计算出流入/流出事件暂存所需的缓冲区大小。通过这种方式，系统的其他特征就可以通过这些组件的信息计算出来。

第二个性能分析方法是由 Henia、Ernst 等人提出的。这种方法被称作 SymTA/S 方法［Henia et al., 2005］, Thiele 的方法中的不同曲线由标准的事件流模型，比如周期性的事件流所取代。周期性的事件流伴随着抖动并且周期性的流也伴随着爆发。SymTA/S 方法明确地支持组合以及集成不同类型的分析技术用于实时研究。

## 5.3　资源与功耗模型

对于相应的系统来说，资源模型以及功耗模型的评估是必不可少的。这两个模型有着紧密的联系，可以从式（3.13）中见到。这些模型旨在通过优化后，降低系统的资源与功耗的损耗。这些方法也试图优化系统的温度，以实现功耗的降低。

第一个功耗模型是由 Tiwari［Tiwari et al., 1994］提出的，这个方法基于观测一个真正的系统。测量值与执行的指令相互关联，该模型内包含了被称作基础损耗和内部指令损耗的概念。每个指令的基础损耗与每条指令执行时消耗的功耗相对应，它假设一个有无穷多个该指令的序列被执行。内部指令损耗模型模拟了处理器发生指令变换时额外的功耗消耗，这些额外的功耗损耗是必需的。例如，切换功能

单元的打开和关闭。这种功耗模型只关注处理器的消耗，并不关心内存或者系统其他部分的功耗。

另一种功耗模型是由 Simunic 等人提出的［Simunic et al.，1999］，这种模型是基于数据表的模型。这种方法的优势是可以使得嵌入式系统的所有组件的功耗都可以计算。但是，数据表中的有关平均值的信息相较于最大值或最小值的信息来说，有可能显得不是特别准确。

第三种模型是由 Rusell 和 Jacome［Rusell and Jacome，1998］提出的，这种模型基于对两个固定的配置进行精准测量。

另外还有一种模型是由 Lee［Lee et al.，2001］提出的，这种模型包括了一种对流水线影响的详细分析。但这种模型没有包括多周期操作以及流水线停滞。

Steinke 等人［Steinke et al.，2001］提出了一种基于实际硬件的精准测量方法，处理器的损耗以及内存的损耗都被包括在内。这种模型被集成进了一种考虑实际能耗的编译器 encc 中，该编译器由 TU Dortmund 公司研发。

通过 CACTI［Wilton and Jouppi，1996］可以对缓存的功耗进行计算。

Wattch 功耗统计工具［Brooks et al.，2000］可以在微处理器系统的架构级别估算功耗，而不需要获取电路设计级别的相关信息。

一些商业工具也提供了功耗统计工具。

功耗估算适用于**功耗管理算法**。

这些示例可以得出如下的结论：对于真实存在的硬件系统来说，可以生成一个精确的功耗模型。然而在设计阶段，由于硬件部分的缺失，功耗模型有可能会不太准确[1]。

## 5.4　热模型

嵌入式系统为了获取更高的性能，往往会使得系统的各个模块在运行时的温度升高。嵌入式系统中各个模块温度的升高，将严重影响系统的可用性。在最糟糕的情况下，过热的组件甚至会导致其他系统的损坏。例如，过热的系统有可能导致火灾的产生。过热的系统组件当然也会导致嵌入式系统自我损毁。然而即使不会对系统产生直接的伤害，过热的部件也会对系统产生其他影响。例如，系统过热会多对系统寿命的降低产生较大的影响。

嵌入式系统的热效应与系统的电—热能转化有着紧密的联系，因此热模型通常与能量模型紧密相连。热模型基于物理原则，首先要考虑**热传导**。热传导反映了一块板子（由某种材料制成），该板子的面积为 $A$ 并且厚度为 $L$。热传导表明了基于 1K 的变化时，所传递的热能量的数量。热导率的倒数被称作**热阻**。对于多块板子来说，总的有效热阻的值是每一块板子的热阻的和。

---

[1] 在讨论中，有时会有大概 50% 的偏差。

这意味着热阻递加的方式有些类似于电路中电阻的叠加。对于热能的贮藏也与电路中电容储电的方式类似。所以，热模型通常等价于电路模型，并且可以使用非常成熟的、用于解决电路网络的方程式（参见例如 Chen 等人的研究 [Chen et al., 2010]）。

热模型的工具包括 HotSpot [Skadron et al., 2009]，该工具可以整合功耗模型，例如 Wattch。这两种工具的集合可以与 SimpleScalar 的功能相提并论 [Simple Scalar LLC, 2004]。热模型的验证需要精准的温度测量 [Mesa-Martinez et al., 2010]。

## 5.5 风险及可靠性分析

嵌入式及物理子系统（如同其他的产品一样）会对人们的生活造成损害。把发生危害的风险降低到 0 是不可能的，所以能做的就是尽可能将损害的严重程度降低，并希望发生该风险的数量级小于其他类型的风险。如今要讨论的任务将在未来变得更加困难，因为随着单位面积二极管数量的增加，二极管设备的可靠性也将随之降低 [ITRS Organization, 2009]。暂态以及永久性的故障将会变得越来越频繁。随着 PCB 板卡面积的减小，也会增加设备参数的不确定性。**因此，可靠性分析以及冗余设计将会变得极其重要** [Mukherjee, 2008]，[Garg and Khatri, 2009]。二极管内部的损坏也许会导致整个系统的崩溃。这些**错误**类型、**故障**的相关术语以及服务是由 Laprie 等人定义的 [Laprie, 1992]，[Avižienis et al., 2004]。

**定义：**

1）“由系统提交的**服务**（角色作为服务提供者）的行为应该能够被用户所感知；递交的服务是服务提供者外部状态的序列；当服务实现了系统功能，该服务就可被认作正确的服务被系统提交”。

2）“一个**有故障的服务**，此处缩写为故障，表明一个系统提供的服务发生了某种事件，导致该服务偏离正确的服务。一个有故障的服务就是从正确的服务到不正确的服务的转变过程。”

3）如果系统中某个状态**不正确**，将导致错误一直存在，并且有可能导致其随后的服务发生故障。

4）“如果某个错误是由判定或假设所导致的，则称之为**故障**。故障有可能来自内部或外部的系统。”

有些故障并不会导致系统的崩溃。

例如，可以假设某个瞬时故障导致了内存比特位的反转。当比特位翻转后，内存单元记录的信息将会发生错误。当系统服务被该错误比特位影响时，将会产生故障。

为了与这些定义一致，将考虑系统在不提供所有这些系统功能的情况下，讨论故障率问题。将在无论假定是否是潜在的原因导致系统故障的前提下，讨论故障。

有无数种原因会导致系统故障的产生，有些故障产生的原因是二极管体积的减小。本书将不会提及有关错误的处理过程。

对许多应用程序来说，发生程序崩溃的概率必须小于$10^{-9}$/h [Kopetz，1997]，这也与每10万个系统运行1万h的场景相对应。达到这种程度的可靠性惟一可行的方法就是在设计评估阶段进行系统的可靠性分析，以达到预期的运行时间和相关目标。这种分析通常基于系统发生故障的概率。

更准确地说，要考虑系统发生故障的概率密度。假定$x$是第一次发生故障的时间，$x$是一个随机的变量，则假定$f(x)$是这个随机变量的概率密度。

作为一个示例，通常使用指数概率密度$f(x)=\lambda e^{-\lambda x}$。对于该密度函数来说，系统的故障有可能随着时间的流动而降低（在未来的某段时间，系统有可能就停止工作，而一个停止工作的系统是无法导致错误产生的）。这种密度函数使用得非常频繁，因为这种密度函数不但有着良好的数学性质，另外在系统实际运行时的故障率往往是未知的。当缺少系统故障率时，会假定一个比率，以用于指数密度函数。指数分布有可能是不准确的，但是它假定了一个典型的至少是粗略近似于真实系统的系统。图5.12左图展示了这个密度函数。

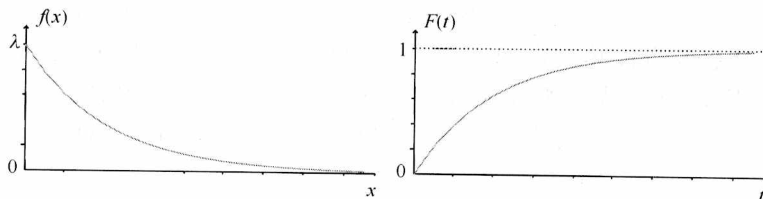



图5.12 指数分布的密度函数和概率分布

概率分布通常来说会比密度更有趣。这个分布表明系统在$t$时刻不工作的概率。该分布可以通过对密度函数到$t$时刻的积分获得：

$$F(t)=\Pr(x\leq t) \tag{5.15}$$

$$F(t)=\int_0^t f(x)\,dx \tag{5.16}$$

例如，通过指数分布可以获得：

$$F(t)=\int_0^t \lambda e^{-\lambda x}dx=-[e^{-\lambda x}]_0^t=1-e^{-\lambda t} \tag{5.17}$$

图5.12右图包含了类似的函数。随着时间的推移，该概率有可能达到1，这意味着随着时间的前进，系统的故障概率将是必然事件。

定义：系统的可靠性$R(t)$是指系统第一次发生故障的概率大于$t$时刻：

$$R(t)=Pr(x>t),t\geq 0 \tag{5.18}$$

$$R(t)=\int_t^\infty f(x)\,dx \tag{5.19}$$

$$F(t)+R(t)=\int_0^t f(x)\,\mathrm{d}x+\int_t^{\infty} f(x)\,\mathrm{d}x=1 \tag{5.20}$$

$$R(t)=1-F(t) \tag{5.21}$$

$$f(x)=\frac{\mathrm{d}R(t)}{\mathrm{d}t} \tag{5.22}$$

对于指数分布，有 $R(t)=\mathrm{e}^{-\lambda t}$（见图 5.13）。

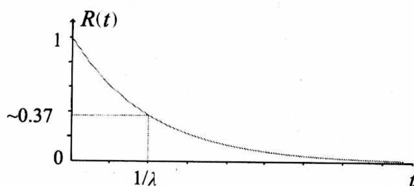



图 5.13　指数分布的可靠性

系统在 $t=1/\lambda$ 时刻发生故障的概率大约是 37%。

**定义：故障率** $\lambda(t)$ 是系统在 $t$ 时刻以及 $t+\Delta t$ 时刻的系统发生故障的概率：

$$\lambda(t)=\lim_{\Delta t\to 0}\frac{\Pr(t<x\leqslant t+\Delta t\mid x>t)}{\Delta t} \tag{5.23}$$

$\Pr(t<x\leqslant t+\Delta t \mid x>t)$ 是系统在 $t$ 时刻时间间隔上的系统崩溃的条件概率。对于条件概率，有一通用的等式 $\Pr(A|B)=\Pr(AB)/\Pr(B)$，$\Pr(AB)$ 是 $A$ 与 $B$ 发生的概率。在本例中，$\Pr(AB)$ 等于 $F(t+\Delta t)-F(t)$。$\Pr(B)$ 是系统在 $t$ 时刻依旧工作的概率，$R(t)$ 作为表示该概率的符号。因此，式（5.23）可以得出：

$$\lambda(t)=\lim_{\Delta t\to 0}\frac{F(t+\Delta t)-F(t)}{\Delta t R(t)}=\frac{f(t)}{R(t)} \tag{5.24}$$

例如，获得的指数分布[⊖]：

$$\lambda(t)=\frac{f(t)}{R(t)}=\frac{\lambda_{\mathrm{e}}^{-\lambda t}}{\mathrm{e}^{-\lambda t}}=\lambda \tag{5.25}$$

故障率经常被观测为 1FIT 的倍数（或分数），“FIT”代表故障发生的时刻。1FIT 对应于每 $10^9$h 发生一次故障。

然而，真正的系统故障率通常是不断变化的，对大多数系统来说，都有一个“浴缸（Bath Tub）”行为（见图 5.14）。

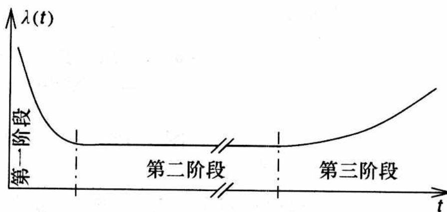



图 5.14　浴缸故障率

对于这种行为，可以发现，在系统的初始阶段的故障率相对更高。导致高的故障率的原因是不完美的生产工艺或“早期

⊖ 该结论促使使用同样的符号表示故障率以及恒定指数分布。

故障期”。在其能够正常工作时，其故障率就变得比较恒定。在产品的最终阶段，由于磨损导致系统故障率又开始变高。

**定义：故障的平均时间**（Mean Time To Failure，MTTF）是从系统初始工作起，直到下次发生故障时的运行平均时间。该平均时间可以由一个随机变量 $x$ 代表的期望值计算得出：

$$\mathrm{MTTF} = E\{x\} = \int_0^\infty x f(x)\,\mathrm{d}x \tag{5.26}$$

该积分可以通过乘积法则计算得出（$\int uv' = uv - \int u'v$，在本例中，有 $u = x$ 以及 $v' = \lambda\,\mathrm{e}^{-\lambda x}$）。因此，由式（5.27）可以得出：

$$\mathrm{MTTF} = -[x\mathrm{e}^{-\lambda x}]_0^\infty + \int_0^\infty \mathrm{e}^{-\lambda x}\mathrm{d}x \tag{5.28}$$

$$= -\frac{1}{\lambda}[\mathrm{e}^{-\lambda x}]_0^\infty = -\frac{1}{\lambda}[0-1] = \frac{1}{\lambda} \tag{5.29}$$

这意味着，对于指数分布来说，直到下次系统故障时的预期执行时间是故障率的倒数。

**定义：平均修复时间**（Mean Time To Repair，MTTR）是假若系统不工作，修复系统的平均时间。该时间是用随机变量的期望值代表系统的修复时间。

**定义：平均故障间隔时间**（Mean Time Between Failures，MTBF）是每两次故障之间的平均时间。

平均故障间隔时间是平均无故障时间及平均故障修复时间之间的和：

$$\mathrm{MTBF} = \mathrm{MTTF} + \mathrm{MTTR} \tag{5.30}$$

图5.15显示了关于该等式的简单示例图，该图并不仅仅反映了处理概率事件的方法，而且显示了实际的平均故障间隔时间、平均无故障时间以及平均故障修复时间的值的变化随机性。

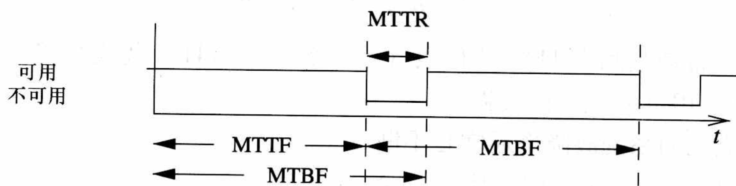



图5.15　图解MTTF、MTTR以及MTBF

对大多数系统来说，并不考虑系统修复的相关事宜。此外，假如考虑系统修复，那么MTTR应明显小于MTTF。因此，MTBF与MTTF经常会混淆不清。例如，硬盘的使用寿命可以引用MTBF，尽管该硬盘也许永远不会被修复。但引用了该参数，则MTTF的时间将会变得更加准确。不过，MTTF仅仅为可靠性提供了非常粗糙的信息，特别是随着时间的变化，故障率也会有很大的变化。

**定义**：可用性是系统处于可运行状态的概率。

系统的可用性也随着时间变化（参考浴缸曲线）。因此，可以通过一个时间依赖函数 $A(t)$ 模拟可用性。然而，时间间隔较大时，更多的是只考虑可用性 $A$。因此，定义：

$$A = \lim_{t\to\infty} A(t) = \frac{\mathrm{MTTF}}{\mathrm{MTBF}} \tag{5.31}$$

例如，假定有一个能够运行 999 天，另外需要 1 天进行修复的系统，那么该系统的可用性 $A = 0.999$。

可以允许的故障率大约是 1FIT。这可能是比芯片的故障率还要低若干数量级的值。这意味着系统必须要比它的组成部分更加可靠！明显地，所需的可靠性的级别意味着就必须有对应的容错技术。

获取实际的故障率是非常困难的。图 5.16 显示了为数不多的公布的实际故障率 [TriQuint Semiconductor Inc., 2010]。

图 5.16 包含了不同的砷化镓（GaAs）设备的二极管在 150℃ 的情况下，运行时的故障率。本例用于说明，无论是假设的恒定故障率还是浴缸行为都是过于简单的。因此，仅仅引用一个 MTTF 数据有可能会产生一定的误导。

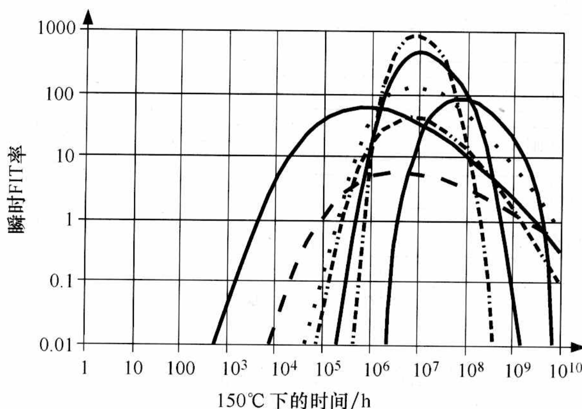



图5.16　TriQuint GaAs 设备故障率（由 TriQuint，Inc.，Hillsboro 提供，© TriQuint）

实际的故障分布随着时间的流逝而被替换。在作为特例的本例中，尽管产品运行于高温中，但在产品生命周期的最初 20 年中（175300h），故障率低于 100 FIT。FIT 的值实际上非常依赖于温度，当温度大于 275℃ 时，Triquint 计算出系统处于失败率的时间已经远远大于系统可用的时间。Triquint 声称，他们的 GaAs 设备相较于其他半导体设备来说，拥有更高的可靠性。关于 FIT 的测试报告也可用于 Xilinx FPGA 中（参见 [Xilinx，2009]）。

对一个完整的系统来说，用实验的方法去验证故障率通常是不可能的。不能使用实验的方法的原因是故障率太低并且故障导致的结果有时无法接受。不能驾驶

$10^5$ 架飞机飞行 $10^4$h，已验证是否已经让系统的故障率低于 $10^{-9}$！解决这种两难境地的惟一出路是检查组件的故障率，从而和使用相结合，由此推导出整个系统的可靠性。设计和用户生成的故障也必须予以考虑。使用决策图来计算系统的组件可靠性，也是一种艺术［Israr and Huss，2008］。

系统的损坏是由潜在的危害导致的（系统故障的原因之一）。对于每一个可能的损坏引起的系统故障，有严重性（成本）和可能性之说。风险也可以被定义为两个产品。关于组件故障导致的损坏，也可以得出有两种对应的技术［Dunn，2002］，［Press，2003］：

1）**故障树分析法**（Fault Tree Analysis，FTA）：FTA 是一个自顶向下的风险分析方法。该分析方法始于一有可能的危害，并尝试提出会导致出现该危害的可能的场景。FTA 基于布尔函数的模型，并反映系统操作的状态（工作或不工作）。FTA 特别包括了与以及或符号，用于代表有可能出现危害的状态。或门用于只要有任意一个系统风险，就会导致系统损害。与门用于表示当有若干个系统风险同时存在时，方可导致系统损害。图 5.17 展示了一个示例[⊖]。FTA 是一个基于系统的结构模型，即它反映了系统组件的划分。

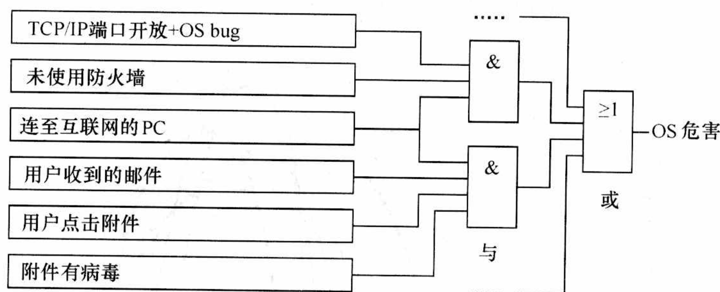



图 5.17　故障树

简单的与门以及或门都不能模拟所有的场景。例如，如果有共享资源或者是一些受限的量存在（例如可用功耗或者存储空间），那么就将超出其建模能力。Markov 模型［Bremaud，1999］或许可以用来处理这些场景。Markov 模型基于状态的概念，而非系统的结构。

2）**故障模式与影响分析**（Failure Mode and Effect Analysis，FMEA）：FMEA 由组件开始进行分析，并尝试去估算其可靠性。利用这些信息，就可以通过对各个组件的可靠性进行计算来获取系统的可靠性（这与自底向上的分析方法相对应）。首先，要创建一个包含各个组件的表，表中包含可能出现的故障、故障的概率以及接下来的系统行为。总的来说，系统风险都可以通过这张表进行计算。图 5.18 显示

⊖ 为了与 ANSI/IEEE 91 标准相一致，使用符号 &，=1 以及 ≥1 来代表与、异或以及或门。

了对应的示例。

| 组件 | 故障 | 结果 | 机率 | 严重? |
|---|---|---|---|---|
| … | … | … | … | … |
| 处理器 | 金属漂移 | 无服务 | $10^{-7}$/h | 是 |
| … | … | … | … | … |

图 5.18　FMEA 表

通过工具的支持，这两种方法都是可以使用的。这两种方法也都可以在“安全情况下”使用。在这种情况下，必须要有一个独立的机构以确保某些技术设备是确实安全的。对技术系统的通常要求是，没有任何一个单独的组件可以导致一场灾难。

对于安全的要求，不应该仅仅做到亡羊补牢，而是要保证整个系统从一开始就是正确的。每一个系统的设计主题都应该是安全且可靠的，本书也只能提供一些有关这些方向的线索。

据 Kopetz［Kopetz，2003］所述，下列事宜应被考虑：对于安全性要求极度苛刻的系统来说，系统的整体必须比其某一部分更加可靠。该系统最多只允许平均每 $10^9$h 出现一次故障。这个数据比芯片发生故障的几率要小 1/1000。因此，容错机制的使用是必需的。由于系统的故障率非常低，所以整个系统有可能没有经过 100% 的验证。但是，系统的安全性必须通过测试和推理相结合进行验证，所以必须使用抽象的方法来解释系统分层的行为模型，设计的失误以及人为造成的故障也必须考虑在内。为了解决这些问题的挑战，Kopetz 提出了如下 12 个设计原则：

1）系统的安全性应该作为系统规格说明书的重要部分，用来推动系统的整体设计。

2）假定设计规格在系统的初始设计时就必须非常精准，规格中包括预期的故障概率。

3）故障控制范围（Fault Containment Regions，FCR）必须被考虑进去，在一个 FCR 中发生的故障不应该影响到其他 FCR。

4）从一开始就应该建立一个统一的时区，否则系统无法辨识初始的及之后的错误。

5）定义良好的接口必须能够隐藏内部的组件。

6）必须确保每个组件的故障不会影响到其他组件。

7）组件应该认为自己是正确的，除非两个或两个以上的其他组件与该组件的预期结果相反（自信原则）。

8）容错机制必须设计为在解释系统的行为时，不会对系统造成额外的负担。容错机制不应该与常规函数挂钩。

9）系统的设计必须经过诊断。例如，它必须能够鉴别出现存在的（但尚未发

现的）错误。

10）人机界面必须直观大方。即使用户的使用有错误，也必须保证系统的安全。

11）应记录每一个异常。有些异常可能无法在常规的接口中观测到。该记录必须涉及内部的影响，否则的话，这些异常有可能被系统的容错机制所掩饰。

12）提供一个永不放弃的策略。嵌入式系统有可能提供不间断的服务。弹出的窗口以及系统的脱线是不可能被接受的。

如果需要有关系统可靠性及安全性的更多信息，请参考书籍［Laprie，1992］，［Neumann，1995］，［Leveson，1995］，［Storey，1996］，［Geffroy and Motet，2002］以及相关领域。

最近有着大量的关于系统可靠性设计类的出版物，例如 Huang［Huang and Xu，2010］，Zhuo［Zhuo et al.，2010］，以及 Pan［Pan et al.，2010］。

## 5.6 仿真

仿真是一种非常普遍的评估以及设计验证技术。仿真一般是在适合的计算机硬件上执行设计模型，特别是在通用数字计算机上。很明显，这需要模型能够被执行。在第2章介绍了所有用于执行的建模语言，这些语言可用于的级别和范围在2.9节做了相应说明。在哪个级别进行仿真设计经常要在速度和精度之间进行妥协。仿真速度越快，那么精度就越低。

到现在为止，使用了术语意义上的系统功能行为（输入/输出行为）。也有一些非功能性的设计行为，包括热行为以及与其他电子设备之间的电磁兼容（EMC）等。由于与物理学的结合，可能有大量的物理效应需要包含在仿真模型内。因此，本书不可能涵盖所有的物理子系统相关的仿真方法。Law［Law，2006］提供了有关数字系统的模拟及仿真的方法综述。

对于物理子系统来说，仿真有着严重的限制：

1）仿真通常要比实际的设计慢很多。因此，如果将仿真结果与实际环境相对应，可能会有相当多的数值违反时间约束。

2）模拟物理环境甚至有可能导致危险的产生（谁想驾驶一辆有着不稳定控制软件的车呢?）。

3）对于许多应用来说，其内部可能需要海量的数据，这可能导致仿真系统其在可用的时间内无法模拟足够的数据。多媒体应用就是其中的一种，例如模拟视频流的压缩，就需要大量的时间。

4）实际的系统往往太复杂，以至于仿真系统无法运行所有的用例（输入）。因此，可以使用仿真系统来帮助人们发现设计错误。仿真系统无法保证系统没有错误，因为仿真系统无法保证去进行所有的系统内部状态及输入的可能组合。

由于这些限制，就要更加注重正式的核查以及验证。然而，先进的仿真技术也将继续发挥其关键作用（例如，Braun et al. [Braunet al.，2010]）。

## 5.7 快速原型及仿真

基于模型的仿真，更加贴近于真实系统。通常来说，真实的系统和模型之间会有一些差异，可以通过实现某些部分的SUD，使得仿真结果更加精确（例如，在一个实际的物理组件中）。

**定义**：采用M^c^Gregor [M^c^Gregor，2002] 的定义，定义**仿真**作为执行SUD模型的过程中，至少有一个组件不在某种主机上进行仿真。

据M^c^Gregor所说，“人们对仿真越来越感兴趣的原因不仅仅是弥合系统和模型之间的差距——上述的有关仿真模型的定义，即使倒转过来也依然有效——仿真模型是一个真实系统的某一部分被模型所取代。”在真实系统下使用仿真模型测试控制系统，通过替换…（真实系统）…为一个模型，对那些需要调试的人或需要自动化安装及启动的系统来说，具有相当大的兴趣。”

为了进一步提高可信性，可以进一步通过真实的组件替换仿真组件。这些组件不必是最终的组件。可以近似于系统本身，但是在精度上必须高于仿真组件。

注意，现在讨论的是在某台计算机上通过软件“仿真”另一台计算机，关于这点一直没有一个明确的定义。然而它可以被认为与定义相一致，因为用于仿真的计算机并不仅仅只用于仿真。相反，它的速度往往高于仿真速度。

**定义**：**快速原型**是执行SUD模型的执行过程，没有任何组件在计算机上被进行仿真。相反，所有的组件都是真实组件，其中的某些组件不应该是最终使用的组件（否则这就是一个真实的系统）。

在许多情况下，在最终的版本制造之前，应尝试在真实的环境中进行系统设计。汽车的控制系统就是一个很好的例子。在大量的产品使用前，应先通过驾驶员在不同环境中的使用，来进行系统研制。因此，这就需要汽车进行工业原型设计。这些原型系统更接近于最终的系统，但是它们有可能更大，消耗更多的功率以及具有其他各种能够被测试的驾驶员所接受的特点。“原型”可以是整个系统，也可以是电气或机械系统。然而快速原型与仿真之间的区别也有些模糊。快速原型的概念也非常宽泛，本书也无法将其全部覆盖。

通过使用FPGA，可以建立和仿真原型。当测试的驾驶员驾驶车辆时，可以将FPGA放在车的行李箱中。这种做法不仅限于汽车行业，还有其他几个使用FPGA建立原型的例子。市场上可以买到的仿真器都包含着大量的FPGA，它们只需要用对应的映射工具去映射对应的规格说明就可以实现对不同系统的仿真。通过使用这些仿真器，最终的试验系统就可以“几乎”像一个真实的系统那样运行。在非分

布式系统中使用仿真或者原型去发现错误是个问题，对于分布式系统来说，这种情况变得更加困难（例如 Tsai [Tsai and Yang，1995]）。

## 5.8　形式验证

形式验证[1]是通过使用数学语言，形式地证明系统的正确性。首先，需要一个可被形式验证可用的形式模型，这一步很难通过自动化来生成，所以此处可能需要较多的工作。一旦确定模型是可用的，就可以尝试去证明某些属性。

形式验证技术可以用逻辑类型进行分类：

1）**命题逻辑**：在这种情况下，模型由布尔函数组成。**这些工具被称作布尔检查器、同意检查器或等值检查器**，它们用来验证两种布尔函数（或布尔函数的集合）是等价的。由于命题逻辑可以判定，它也可以用于判定两种表达式是否等价（这是毫无疑问的）。例如，一个表达式既可以与实际电路中的门电路对应，也可以与其他规范对应。当提供了设计类似的表达式后，需要通过等值检查器去检测设计（例如，电源优化）是否正确。布尔检查器可以应对大多数的，基于仿真的详细验证。布尔检查器功能强大的原因是由于使用了二叉决策图（Binary Decision Diagram，BDD）[Wegener，2000]。以二叉决策图来表示布尔函数的等值检查器的复杂度，与二叉决策图的节点数量成线性关系。与之相反，用于等值检查的函数由大量的非确定性多项式来表示。因此，基于二叉决策图的等值检查器可以替代仿真器用于拥有数以百万计二极管的电路。

2）**一阶逻辑**（First Order Logic，FOL）：FOL 包括 ∃ 以及 ∀ 操作符。通常情况下，整形数据可以进行运算。这样，自动化验证 FOL 模型就是可行的。然而，由于 FOL 不可判定，在某些情况下，有一定的不确定性。**霍尔微积分**就是一种常用的 FOL 技术。

**高阶逻辑**（Higher Order Logic，HOL）：高阶基于 λ 演算并且可以像其他的对象一样操作函数 [University of Cambridge，2010]。高阶逻辑已经被证明不可能用自动化的方式进行验算，并且通常需要手动进行验算支持。

命题逻辑可以用来验证无状态的逻辑网络，但不能直接模拟有限状态机。对于短输入序列，它可能足以削减有限状态机中的反馈环，并且可以有效地处理这些有限状态机的副本，每一个副本都代表着一个输入类型的影响。然而，这种方法不适合用于较长的输入序列。这种序列可以使用模型校验进行处理。

对于模型校验，有如下两种输入验证工具：

1）模型验证；

[1] 形式验证部分的说明来自于客座教授 Tiziana Margaria-Steffen（来自多特蒙德理工大学）。

2）属性验证。

状态可以通过无法用数字量化的∃以及∀符号量化。验证工具可以用于证明或者反证属性。对于后者，可以提供一个反例。模型验证相较于FOL更容易实现自动化。它在1987年通过使用BDD首次实施。它有可能在总线的协议规范中定位若干的错误［Clarke et al.，2005］。

下一步将会尝试整合模型检查器以及高阶逻辑。在这种集成模型中，高阶逻辑只在绝对需要时才会被使用。

Clarke的EMC系统［Clarke and et al.，2003］（见图5.19）可以作为该方法的一个示例。

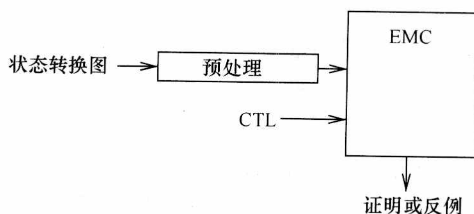


图5.19　Clarke的EMC系统

该系统尝试使用CTL公式描述系统。CTL公式包括两个部分：

1）**路径计算器**（这部分指定路径中的状态转换图）；

2）**状态计数器**（这部分指定状态）。

例如：$M, s \models AGg$ 意味着：在转换图 $M$ 中，有一个适用于所有路径（由 $A$ 表示）的属性 $g$ 开始时的状态与所有状态（由 $G$ 表示）。可以延伸用于表达覆盖实时及数字行为。

这种技术还可用于，如证明铁路属性的模型，如图2.52所示。它可以转换为状态图的Petri网，然后确认巴黎以及科隆之间的列车数量是恒定的，这证实了关于2.6.3节讨论的Petri网的不变量。

## 5.9　思考题

1. 考虑一个有关帕累托最优的概念的例子。在本例中，学习由IMEC研究中心设计的并发任务管理（TCM）工具。TCM工具用于在应用程序与处理器间进行有效映射。不同的多处理器系统评估，用帕累托最优设计表示。Wong等人［Wong et al.，2001］描述了不同的MEPG-4播放器的设计方案。作者在此处假定使用StrongARM处理器以及特别的加速器的组合。该设计应满足30ms的时间约束（见图5.20）。

| 处理器组合 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 高速处理器数量 | 6 | 5 | 4 | 3 |
| 低速处理器数量 | 0 | 3 | 5 | 7 |
| 总处理器数量 | 6 | 8 | 9 | 10 |

图5.20　处理器配置

这些不同的设计如图5.21所示。

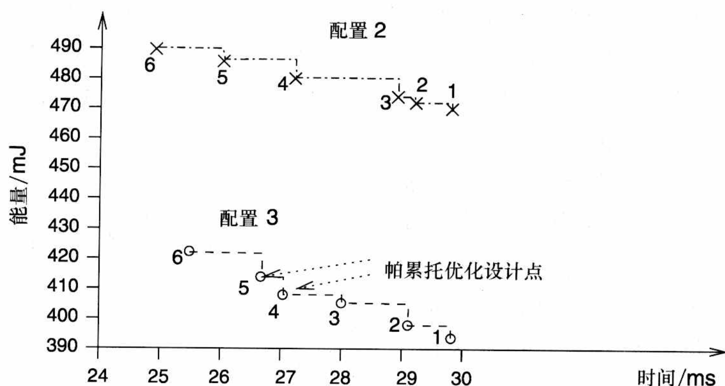



图5.21　多核处理器系统配置2以及配置3的帕累托点

对于组合1和4，作者报告中提到，只有一个处理器到任务之间的映射满足时间约束。

对于组合2和3，不同时间分配不同的任务导致不同的处理器映射以及不同的系统功耗。

哪一片目标空间至少由一个配置3所控制？有哪些设计属于不由配置3支配的配置2？目标空间的哪片区域控制着至少一个配置3的设计？

2. 什么情况下必须满足 $WCET_{EST}$ 的计算？

3. 考虑控制流加入后高速缓存的状态！图5.22显示了加入控制流之前的高速缓存状态。现在看一下加入控制流后的缓存状态，哪种状态是必须分析得出的？那种状态是可能分析得出的？

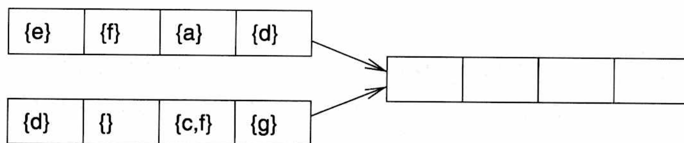



图5.22　抽象缓存状态

4. 考虑传入的“突发”事件流。该流是以 $p$ 为周期的周期性流。在每个周期开始时，每两个事件的到达都以 $d$ 作为时间单元进行间隔。优化该流的到达曲线！最终的图形应显示为 $0 \sim 3 * p$。

5. 假设有一个所能工作的最高性能为 $b$ 的处理器。

1）如果系统的服务曲线看起来由于缓存的冲突导致系统性能降低到了 $b$，怎么办？

2）如果每100ms就有时钟中断打断正在运行的程序并且每次时钟中断的中断

处理程序消耗时间为 10ms，那么系统的服务曲线该如何改变？假设此处没有缓存冲突。

3）在考虑类似 1）中的缓存和类似 2）中的中端情况下，服务曲线是什么样的？

结果图需显示 0 ~ 300ms 的时间段。

# 第 6 章　应用程序映射

## 6.1　问题定义

一旦系统规格已经定义完毕，就可以开始设计工作了。这与简化了的设计信息流是一致的（见图 6.1）。向要运行的平台上进行程序映射确实非常关键，因此在本章中将特别予以强调。

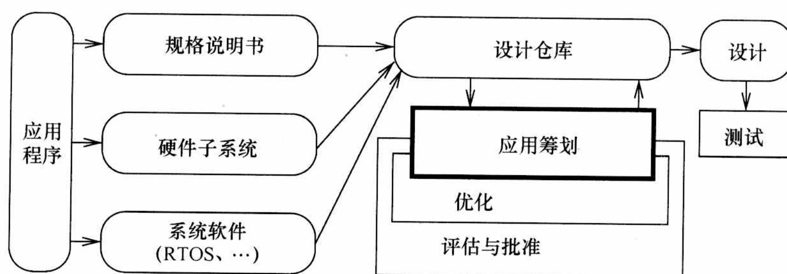



图 6.1　简单设计流

对于嵌入式系统来说，通常希望系统上运行某些特定应用程序的组合。例如对于移动电话来说，希望可以在打电话的同时，使用蓝牙将对应的音频信号传递到耳机中，并且同时查看“个人信息管理（Personal Information Manager，PIM）”去查找某些信息。期望在同一时间可以并发的进行文件传输，甚至是视频连接。必须确保这些应用程序可以同时工作并且还能保证其死线（不丢失音频采样!）。通过用例分析，可以确保上述场景是可行的。

这也是嵌入式物理子系统的一个特点，就是在设计阶段就要同时考虑软件和硬件，因此这种类型的设计也被称作软、硬件协同设计。总的目标就是尽可能找出最合适的软、硬件组合的方式，最快捷、有效地满足产品设计规范。因此，嵌入式系统不能仅仅是只简单地将各个进程按照规范进行组合，相反每一个用到的组件，都必须说明引用该组件的原因。当然，也有其他原因导致该约束的产生，那就是为了满足嵌入式系统日益增长的复杂度以及迫切的市场需求，重用从本质上来说就是不可避免的。由此得出了**基于平台的设计原则**。

“平台就是一个满足限制的、可重用的软件模块和硬件模块的集合，然而仅有硬件平台是远远不够的。快速、可靠以及扩展的设计，需要一个通用的平台 API 用于平台的应用软件。一般来说，平台是一个在底层可以覆盖大多数可能用到的功能

的抽象层。基于平台的设计是一种折中的方法：在自顶向下的设计过程中，设计师将一个位于上层平台的实例映射到底层平台上，并且进行一些设计上的约束。”[Sangiovanni-Vincentelli，2002]。映射就是在性能评估工具引导下的迭代过程。

在本书中，关注于可供执行平台的嵌入式系统设计，这也反映了许多现代系统是建立在某些已经存在的平台上的事实。本书描述的技术是基于可执行平台的设计，并且已被实际应用。鉴于人们的关注点，应用程序在**可执行平台上的映射**可以被看作**主要的设计问题**。

映射的执行通常基于多核处理器系统，一般而言有如下两种多核处理器系统：

1）**同构多处理器系统**：在这种情况下，系统中的所有处理器都提供同样的功能，这是类似于 PC 系统的多核处理器架构。不同处理器之间的代码兼容性是其主要优势：它可以在不同的核之间进行任务的调度（包括负载均衡），并且在系统容错设计上也有不少优势。如果某个处理器发生故障，可以将进程重新分配到其他处理器上运行。此外，如果所有处理器的类型都是相同的，那么处理器平台的开发设计工具的使用就会变得更加简单、方便。

2）**异构多处理器系统**：在这种情况下，有多种类型的处理器。使用这种方法的关键原因是这种方法有同构处理器系统所达不到的性能优势。异构多处理器系统是最高效的编程平台。

即使是基于平台的设计，可能也有着若干种设计方案。有可能选择一个平台的不同变种，例如某个变种有着不同数量的处理器、某个变种有着不同速度的处理器或者不同的通信架构。此外，适用的调度策略也会有着不同的变化。只能在多种方案中选择恰当的一种。

这使得对映射的问题定义如下 [Thiele，2006a]：

给定：

1）一组应用程序；

2）用于描述应用程序该怎样使用的使用案例；

3）一组可能的候选架构：

①（可能是异构的）处理器，

②（可能是异构的）通信架构，

③ 可能的调度策略。

找出：

1）应用程序在处理器上的映射；

2）合适的调度技术（如果没有准备好）；

3）目标架构（如果没有准备好）。

宗旨：

1）保证系统死线以及最佳性能；

2）要降低系统成本、系统功耗以及其他可能达到的目标。

寻找可能的架构方案被称作设计空间探索（Design Space Exploration，DSE），一个完全固定的平台架构可以被看作一个特例。

基于AUTOSAR设计的汽车系统可以被用作一个示例：在AUTOSAR［AUTOSAR，2010］中，有若干同质执行单元（被称作ECU）以及若干软件构件。问题就是，如何在满足实时性约束的前提下，使用最低数量的ECU，并将软件构件映射到ECU上。

应用程序的映射是一个非常难解决的问题，应用程序的自动映射还没有实现。接下来，将展示一些用于映射的模块：

1）标准调度技术；

2）硬件/软件分区；

3）将一组应用映射到多处理器系统上的技术。

下面将介绍可以用于不同情况下的标准调度技术的说明。

## 6.2　实时系统中的调度

如前面所述，调度是实现嵌入式系统的关键因素之一。在设计系统时，有可能要对调度算法进行若干次调整。系统规格一旦确定后，对于调度算法的使用就有了一个大概的估算。接下来，可能需要更加详细的有关执行时间的预测。在代码编译后，可以获得更详细的有关执行时间的细节以及更精准的调度规划，最终决定后续任务该如何被执行。但与之相反的是，在基于时间触发的系统中，RTOS的调度器仅仅通过简单的查表来选择哪个进程被调度。调度也像进行性能评估一样，并不能仅仅在设计的某一个步骤中就决定使用何种调度方式。

调度器定义了每个任务的开始时间，并且定义了从$\tau$到每个任务图$G=(V,E)$到时域$D_t$的节点的映射：

$$\tau: V \rightarrow D_t \tag{6.1}$$

### 6.2.1　调度算法分类

调度算法可以根据不同的标准进行分类。图6.2显示了其中的一种分类方法（类似的方法在［Balarin et al.，1998］，［Kwok and Ahmad，1999］，［Stankovic et al.，1998］，［Liu，2000］，［Buttazzo，2002］的书籍中各有讨论）。

接下来是一个有关分类标准的讲述，前4种方式都与图6.2相关：

1）**软死线以及硬死线**：基于软死线的调度算法通常是在标准操作系统上进行扩展。例如，提供任务所需的系统调用的优先级的确定，使用基于软死线的系统就足矣，本书不会更进一步讨论此类系统，本书将会把更多的精力放在硬死线系统上，这些问题将在周期系统和非周期系统上讨论。

2）**基于周期和非周期的任务调度**：接下来，首先要分辨什么是周期任务，什

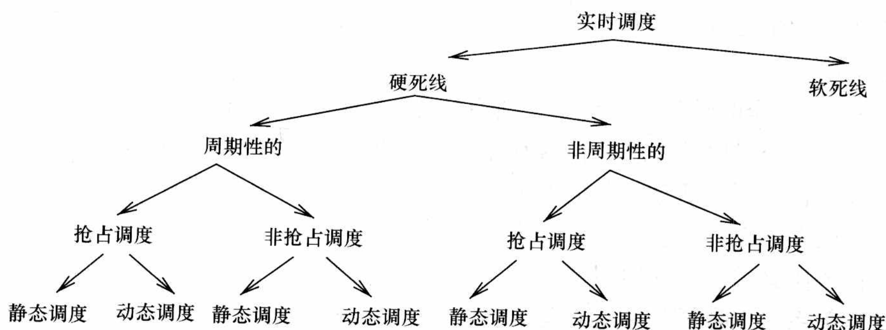



图 6.2 调度算法分类

么是非周期任务。

**定义**：在每个以 $p$ 为单元时间内，都要执行一次的任务，叫做**周期性任务**。$p$ 被称为是这些任务的**周期**。每一个周期任务的执行过程被称作**工作**。

**定义**：不以固定周期运行的任务被称作**非周期任务**。

**定义**：如果在申请处理器时有一个最小间隔时间，那么非周期性任务在不可预知的时间进行处理器请求被称作**突发请求**。

这个最小间隔非常重要，因为如果程序组没有这个最小间隔时间，则无法进行进程调度。因为如果任务一旦变得可执行，那么有可能没有足够的时间来进行任务的执行。

3）**抢占以及非抢占调度**：非抢占调度基于程序能够一直运行到程序自我执行完毕的假设。因此，如果某些任务执行时间过长，那么对于某些外部事件[1]的响应时间可能会变得相当长。如果某些任务需要较长的运行时间，或者系统要求对外部事件的响应时间需尽可能短，那么就需要使用抢占式调度。然而，抢占式调度有可能导致系统的运行时间变得不可预测。因此，对于某些硬实时任务来说，要限制抢占，以保证硬实时任务能够在死线到来前结束运行。

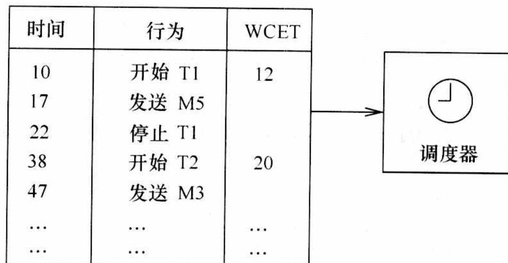
4）**静态与动态调度**：动态调度在任务运行时决定调度策略。这种调度方式非常灵活，但是会在运行时产生一定的系统开销。此外，该方式通常不关注全局的系统上下文，例如资源需求或任务间的依赖。对嵌入式系统来说，这种全局的上下文在系统设计时就应该被考虑。

静态调度在设计时就应决定该如何调度。此种调度方法基于规划好任务的起始时间，并生成一个任务起始时间表，交付到一个简单的调度器中。调度器并不做任何调度决策，仅仅是根据调度表中指明的任务起始时间开始任务运行。调度器可以由时钟进行控制，并进行调度表的分析。完全由时钟控制的系统被称作**完全时钟触**

[1] 这是从外部事件发生到外部事件处理结束的响应时间。

发系统(Time Triggered，TT）系统。此类系统在 Kopetz［Kopetz，1997］的书籍中有详细论述：

“在一个完全时钟触发系统中，所有任务的时序控制结构首先由某种离线支持工具建立。这个时序控制结构包含在了**任务描述符表**（Task-Descriptor List，TDL）中，该表包含了所有活动任务的调度周期⊖（见图 6.3）。这个调度表考虑到了任务所必需的优先级以及任务间的互斥关系，以致操作系统在运行时对任务调度的协调工作是完全不必要的”。图 6.3 包括了调度器中任务开始、任务结束以及消息发送的过程。



| 时间 | 行为 | WCET |
| --- | --- | --- |
| 10 | 开始 T1 | 12 |
| 17 | 发送 M5 | |
| 22 | 停止 T1 | |
| 38 | 开始 T2 | 20 |
| 47 | 发送 M3 | |
| … | … | … |
| … | … | … |



图 6.3　时间触发系统中的 TDL

“调度器由同步的时钟滴答激活。激活后首先观察 TDL，并根据 TDL 的表项立即执行原定的调度计划…”。

静态调度器的主要优点是，如果系统满足原定的时间约束，那么系统很容易进行检查：

“为了满足硬实时系统的时间约束，系统的可预测性就变得更加重要；相对于复杂系统而言，预运行时间调度往往是提供系统可预见性的惟一手段”［Xu and Parnas，1993］。

静态调度器的主要不足是对于突发事件的反映显得缺乏手段。

5）**独立任务以及相关任务**：区分某个任务是否与其他任务有进程间的通信是可能的。对嵌入式系统来说，任务之间经常会有依赖关系，没有依赖关系的任务往往是种例外。

6）**单处理器调度与多处理器调度**：简单的调度算法通常用于单处理器系统，反之复杂的调度算法用于处理多处理器组成的系统。对于后者，可以区分同质多处理器算法以及异构多处理器算法。异构多处理器算法可以有针对性地处理执行时间以及用于混合的软/硬件系统，并将其中的某些任务映射到硬件。

7）**集中式调度以及分布式调度**：多处理器调度算法既可以在某个局部处理器上进行调度，也可以分布到多个处理器上进行调度。

8）**可调度性类型以及复杂度测试**：实际上，是否有一个满足给定的一组任务限制的调度器是非常重要的。

如果有一组任务以及对应的限制规则存在的情况下，还存在一个调度表，那么就说这组任务在给定的限制下是可调度的。对于许多应用程序来说，可调度性测试

⊖　这个术语用于在此场景下的处理器。

非常重要。在许多情况下，使用非确定性多项式可以返回一个精确的（被称作确切概率法）测试结果［Garey and Johnson，1979］。因此充分性以及必要性的测试，都要用该多项式进行测试。对于充分性测试，要保证调度器进行充分的状态检查。检查结论要指明即使有调度表存在的情况下，调度器依旧不能进行调度的概率（希望这个值尽可能小）。必要性测试基于检查系统的必要状态，这种测试可以在没有调度表的情况下进行验证。所以，即使调度表不存在，必要性验证依然有可能通过。

9）**成本函数**：不同的算法旨在最大限度地减小函数的规模，最大延迟通常用于成本函数中。

**定义**：最大延迟定义为所有任务的完成时间与死线之间的最大差值。如果所有任务在它们的死线之前执行完毕，那么最大延迟的值为负。

## 6.2.2　没有优先级约束的非周期性调度

### 6.2.2.1　定义

假如 $\{T_i\}$ 是一组任务，让（见图 6.4）：

1）$c_i$ 是 $T_i$ 的执行时间；

2）$d_i$ 是死线的间隔时间，换言之，直到 $T_i$ 结束运行之前，$T_i$ 之间的时间都是有效的。

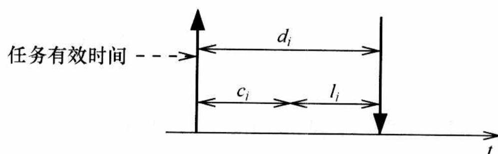



图 6.4　闲置任务定义

3）$l_i$ 是闲置的时间，定义为

$$l_i = d_i - c_i \tag{6.2}$$

此外，向上的箭头指明任务开始有效的时间，向下的箭头指明了任务的死线。

如果 $l_i = 0$，那么 $T_i$ 在其变为可执行状态时，要立即执行。

首先考虑[㊀]在单处理器系统中，所有任务同时到达的情况，如果所有任务同时到达，那么抢占就变得没有意义。

### 6.2.2.2　最早交付日期算法

Jackson 在 1955 年［Jackson，1955］首先提出了最早交付日期（Earliest Due Date，EDD）算法这种简单的调度算法，该算法基于 Jackson 提出的规则：

该算法是给定一组 $n$ 个独立任务，在没有达到死线的所有任务中，选择一个距离死线最近的任务执行的算法。

遵守这种规则的算法被称作 EDD 算法。如果能够提前知道任务死线，EDD 算

㊀ 这里使用了 Buttazzo［Buttazzo，2002］所著书籍中的某些部分来阐述此部分。如想获得额外的参考，可以查阅此书。

法可以用作一种静态调度算法。EDD 算法需要所有的任务根据其死线进行排序，因此其算法复杂度为 $O(n\log(n))$。

EDD 系统的性能验证如下：

让 $\tau$ 作为由算法 $A$ 生成的调度表。假设 $A$ 得出的结论与 EDD 不同。接下来，调度表中有 $T_a$ 与 $T_b$ 两个任务，尽管 $T_a$ 的死线先于 $T_b(d_a<d_b)$，但 $T_b$ 在 $T_a$ 之前运行。现在考虑调度表 $\tau'$，$\tau'$ 由交换 $\tau$ 中的 $T_a$ 以及 $T_b$ 的执行顺序得出（见图 6.5）。

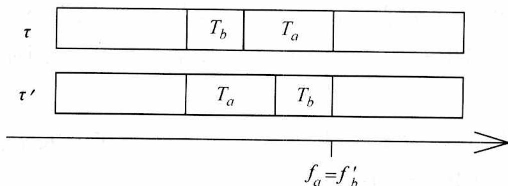



图 6.5　调度表 $\tau$ 以及 $\tau'$

$L_{\max}(a,\ b)=f_a-d_a$ 是调度表 $\tau$ 中任务 $T_a$ 与 $T_b$ 的最大延迟。对于调度表 $\tau'$ 来说，$L'_{\max}(a,\ b)=\max(L'_a,\ L'_b)$ 是任务 $T_a$ 与 $T_b$ 中有最大延迟的一个。$L'_a$ 是任务 $T_a$ 在调度表 $\tau'$ 中的最大延迟，$L'_b$ 的定义与之相符。有如下两种可能的情况：

1）$L'_a>L'_b$，在这种情况下有

$$L'_{\max}(a,b)=f'_a-d_a$$

$T_a$ 在新的调度表中较早停止，因此有

$$L'_{\max}(a,b)=f'_a-d_a<f_a-d_a$$

不等式的右侧是调度表 $\tau$ 中的最大延迟，因此接下来有

$$L'_{\max}(a,b)<L_{\max}(a,b)$$

2）$L'_a\leqslant L'_b$，在这种情况下有

$$L'_{\max}(a,b)=f'_b-d_b=f_a-d_b\text{（见图 6.5）}$$

$T_a$ 的死线比 $T_b$ 提前到达，这导致：

$$L'_{\max}(a,b)<f_a-d_a$$

接下来有

$$L'_{\max}(a,b)<L_{\max}(a,b)$$

作为结论，任何调度器（即使不是 EDD 调度器），也可以通过对有限的任务转换实现 EDD 调度。最大延迟只可以通过任务的转换降低。因此，EDD 是所有调度算法中的最优算法。

#### 6.2.2.3　最早到期优先算法

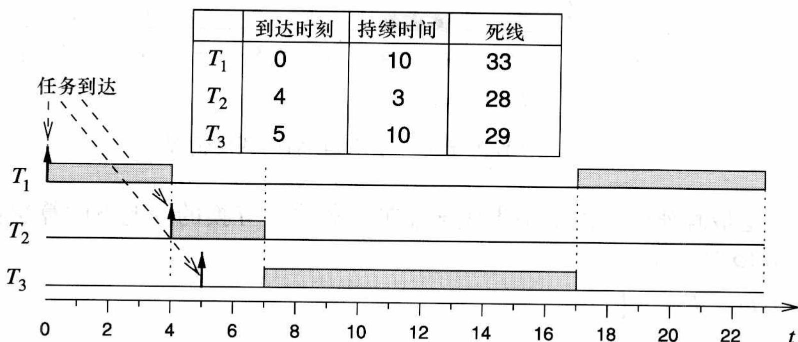
下面考虑在单处理器系统下，不同的任务到达时间的情况。在这种情况下，抢占有可能减少最大延迟。

最早到期优先（Earliest Deadline First，EDF）算法可以优化最大延迟时间。该算法基于如下原理［Horn，1974］：

给定一组具有任意到达时间的 $n$ 个独立任务，任何具有最早绝对死线的，已经准备好立刻可以执行的任务，无论是由何种算法，其最终意义都是优化器最大延迟。

EDF 要做的是，每当有一个准备就绪的任务到来时，都根据其死线排序，并将其插入一个任务的就绪队列中，因此 EDF 是一个动态调度算法。如果一个新到来的任务被排在了就绪队列的首部，那么当前正在执行的任务就会被**抢占**。如果队列使用一个排序表，那么 EDF 的时间复杂度为 $O(n^2)$。Bucket 数组可以用于减少执行时间。

图 6.6 显示了由 EDF 算法得出的调度表，垂直的箭头指示了到达的任务。



|  | 到达时刻 | 持续时间 | 死线 |
|---|---|---|---|
| $T_1$ | 0 | 10 | 33 |
| $T_2$ | 4 | 3 | 28 |
| $T_3$ | 5 | 10 | 29 |



图 6.6　EDF 调度

在时刻 4，任务 $T_2$ 有一个较早的死线。因此 $T_2$ 抢占 $T_1$。在时刻 5，任务 $T_3$ 到达。由于其死线较晚，所以 $T_3$ 不会抢占 $T_2$。

**最优 EDF 证明如下：**

$\tau$ 是由算法 $A$ 所得出的调度表，算法 $A$ 不基于 EDF 算法。调度表 $\tau_{\mathrm{EDF}}$ 基于 EDF 算法。现在将时间分割为以长度 1 为单位的不相交的区间，每一段时间间隔都由区间 $[t, t+1)$ 组成。让 $\tau(t)$ 作为按照预定调度计划 $\tau$ 执行的任务，在 $[t, t+1)$ 的间隔内运行。让 $E(t)$ 作为所有任务中具有最早死线时间的任务，让 $t_E(t)$ 作为任务 $E(t)$ 在 $\tau$ 调度表中的开始执行时间（$\geq t$）。

$\tau$ 并不是 EDF 调度器，因此在 $t$ 时刻必然有一个具有最早死线的任务停止执行。对于 $t$，有 $\tau(t) \neq E(t)$（见图 6.7）。死线由向下的箭头表示。

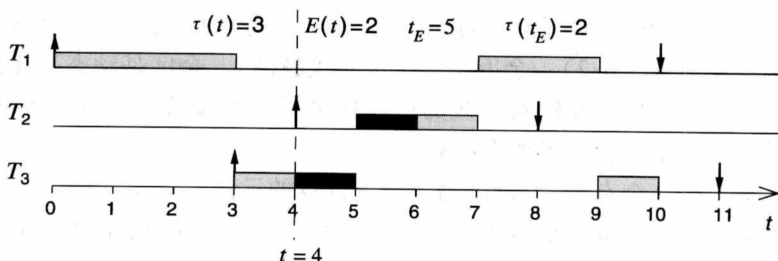




图 6.7　调度表 $\tau$

证明的基础思路是显示对 $\tau(t)$ 与 $E(t)$ 的交换（见图 6.8）不会增加最大延迟。

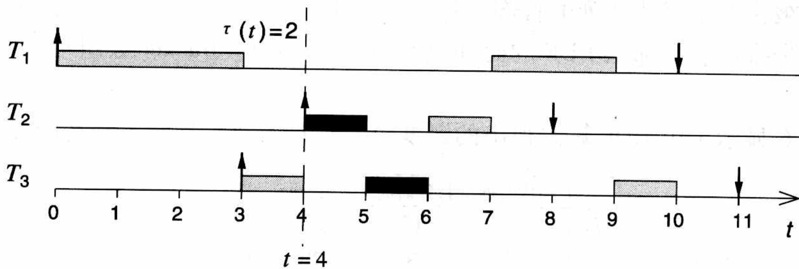



图 6.8 互换任务 $\tau(t)$ 与 $E(t)$ 后的调度情况

假设 $D$ 是最晚死线，由 $\tau$ 中生成 $\tau_{EDF}$ 在大多数 $D$ 互换的情况下的算法：

```
for (t=0 to D-1) {
  if (τ(t) ≠ E(t)) {
    τ(t_E) = τ(t);
    τ(t) = E(t); }}
```

通过使用与 Jackson 规则相同的参数可以得知，任务的互换不会增加最大延迟。因此，任何非 EDF 调度器都可以在不增加最大延迟的情况下转化为 EDF 调度器。这意味着 EDF 是所有可用的调度算法中，最优的一种。倘若这些任务仍旧在调度表 $\tau$ 中，那么可以得知，任务的互换不会导致其死线产生任何变化。首先考虑任务 $E(t)$，该任务将会比之前的调度表中的时间更早的运行，因此如果该任务在原有的调度表中有死线，它也会在新的调度表中遇到死线。接下来考虑任务 $\tau(t)$，$\tau(t)$的死线比 $E(t)$ 更大，因此如果 $E(t)$ 在旧的调度表中有死线存在，那么 $\tau(t)$ 也会在新调度表中遇到死线。

#### 6.2.2.4 最小余度算法

最小余度（Least Laxity，LL）、最少时间轮转优先（Least Slack Time first，LST）以及最小余度优先（Minimum Laxity First，MLF）是另一种调度策略的三个不同名称［Liu，2000］，对于 LL 调度算法来说，任务的优先级是一个基于余度单调递减的函数［见式（6.2），余度越小，优先级越高］。余度是动态变化的，所以也需要动态的计算。负余度针对死线的错过提供了一个预警。LL 调度器也支持抢占，抢占并不限定于新任务可调度的时间点上。

图 6.9 显示了一个有关 LL 调度器与余度计算的示例。

在时间点 4 上，任务一被抢占。在时间点 5 上，由于 $T_3$ 的余度最小，$T_2$ 也被抢占。

可见（这是一个被用作思考题的示例［Liu，2000］），对单处理器系统来说，在此种场景下，LL 调度器也是一个理想调度策略。对于其动态优先级而言，该调

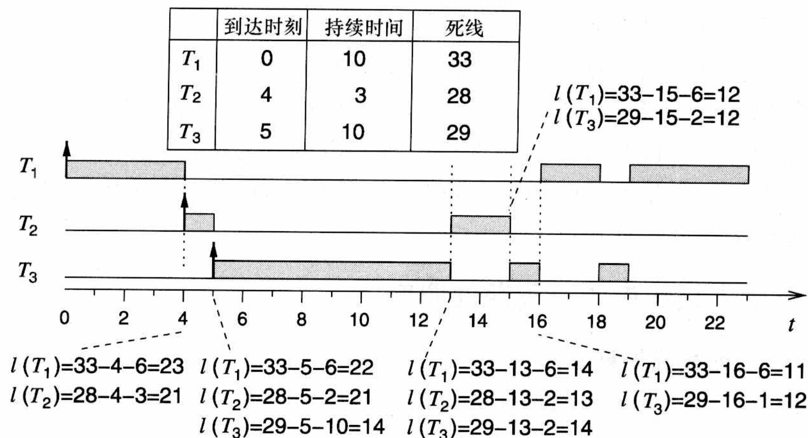



图 6. 9　LL 调度器

度器不能用于只提供固定优先级的标准操作系统。此外，与 EDF 调度器相比，LL 调度器要知道任务的执行时间，因此 LL 调度器被限制在某些对性能要求较高的特殊情况下使用。

#### 6. 2. 2. 5　非抢占式调度

如果不允许任务抢占，那么理想的调度器必须在某时刻能够结束任务，以保证任务不会越过死线时间。

**证明：**假设一个不支持抢占的理想调度器，该调度器即使在处理器空闲时也不会停止工作。该调度器必须用合适的方式调度图 6. 10 所示的示例（如果有调度表存在，那么调度器必须依照调度表进行调度）。

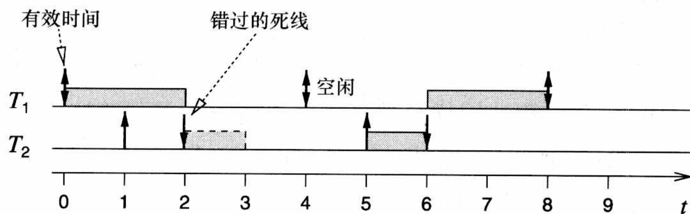



图 6. 10　需要离开处理器闲置状态的调度器

对于图 6. 10 所示的示例，假设有两个给定的任务。让 $T_1$ 作为执行时间为 2 的周期任务，该任务的周期为 4，并且死线的间隔也为 4。$T_2$ 是一个偶尔运行的任务，该任务在 $4*n+1$ 时可执行，并且执行时间以及死线的间隔为 1。假设 $T_1$ 和 $T_2$ 不可能并发执行（例如，使用了一个单处理器），基于上述假设，调度器在时间点 0 开始执行任务 $T_1$。由于调度器不允许抢占，因此在时间点 1 时，$T_2$ 没有运行，因此 $T_2$ 也会漏掉它的死线。如果调度器在处理器空闲时停止工作（见图 6. 10 在时间点 4），那么一个规定好的调度器开始运行，因此该调度器不是最优的。最优的

调度器在处理器空闲时不会停止工作是一个矛盾的假设。

推断：为了避免丢失死线，调度器必须知道任务的调度情况。如果调度器对任务到来时间以及优先级一无所知，那么调度算法就不知道是否保持处理器的空闲状态还是运行状态。可以得知，在不保持处理器空闲状态的前提下，EDF 是所有算法中的最优算法。如果到达时间是先验的，调度问题就成为了使用非确定性多项式以及分支及边界技术生成调度表。

## 6.2.3　有优先级约束的非周期性调度

### 6.2.3.1　最近死线优先算法

下面通过一个任务图来反映任务间的依赖关系（见图 6.11）。任务 $T_3$ 只有在 $T_1$ 以及 $T_2$ 执行完毕，并发送信号给 $T_3$ 后，方可执行。

图 6.11 也显示了一个合法的调度表。对于静态调度来说，该调度表可以存储在一个表格中，指明任务的开始以及信息交换的时间。

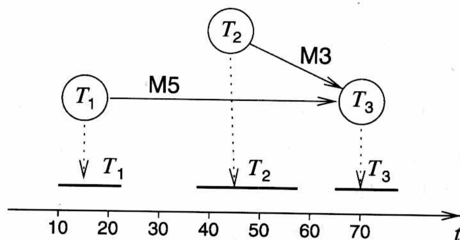


图 6.11　优先图以及调度表

Lawler［Lawler，1973］提出了一个能够同步到达时间，以实现最小化最大延迟的最优算法，该算法被称作**最近死线优先**（Latest Deadline First，LDF）算法。LDF 读取任务表，并向队列中插入没有继承者的任务。LDF 接下来重复这个过程，并将有继承者的任务插入该队列。在运行时，任务以插入队列的相反顺序执行。LDF 算法是不支持抢占的单处理器下的最优算法。

可以用改进的 EDF 算法处理异步到达任务下的情况。关键的思路是从一组给定的、具有依赖关系的任务转变为一组有不同的、时间参数的、不相关的任务［Chetto et al.，1990］。该算法同样是单处理器系统下的最优算法。

如果是基于**非抢占**的情况，也可以使用 Stankovic 以及 Ramamritham［Stankovic and Ramamritham，1991］提出的启发式算法。

### 6.2.3.2　最快调度

有些致力于编程社区也提出了另外的调度算法，如最快（As-Soon-As-Possible，ASAP）调度、最晚（As-Late-As-Possible，ALAP）调度、列表（List，LS）调度，以及在做上层集成（High Level Synthesis，HLS）社区中非常流行的力向调度（Force-Directed Scheduling，FDS）算法（见［Coussy and Morawiec，2008］HLS 最近成果），ASAP 以及 ALAP 调度不考虑任何资源或时间约束。FDS 考虑全局时间约束，LS 要考虑资源约束。

下面将用一个简单的表达式来证明前三种算法，考虑到一个 3×3 矩阵（见图 6.12）。

$$\boldsymbol{A}=\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$$

图 6.12　3×3 矩阵

该矩阵的行列式 det(*A*) 可被计算为

$$\det(A) = a*(e*i-f*h)+b*(f*g-d*i)+c*(d*h-e*g)$$

该计算过程可以由一个数据流图表示（见图 6.13）。假设每个算数计算都代表了一个简单的“任务”。

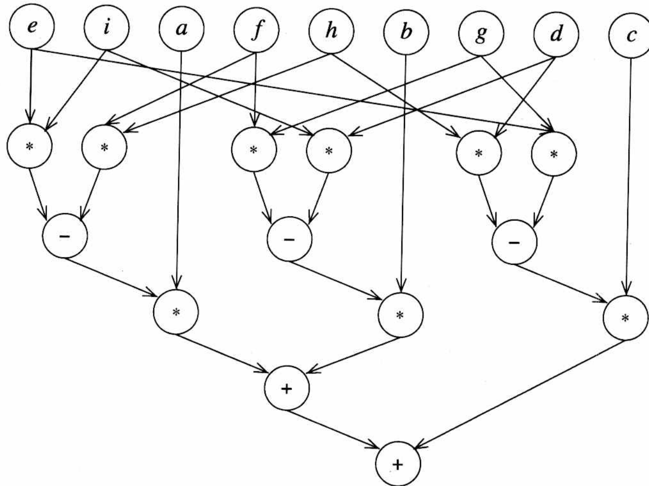



图 6.13　行列式 *A* 的计算

约定所有的矩阵值都是可以立即使用的（例如，这些值都存放在了寄存器中）。

HLS 在使用 ASAP 时，会认为任务映射为一个起始时间[㊀] >0 的整数，因此调度器提供了一组映射：

$$\tau: V \rightarrow IN \tag{6.3}$$

这里 $G=(V, E)$ 是数据流图。

对于 ASAP 调度，所有的任务都尽可能早开始运行，算法工作流程如下：

```
for (t=1; 所有任务都被调度; t++) {
  s={所有任务的所有输入者是使能的 };
  设置所有任务t的起始时间为s;
}
```

为了简单起见，假设例子中所有加法以及减法的执行时间都为 1，而乘法的执行时间为 2。图 6.14 显示了基于示例的调度器的数据流图 6.13 的结论。

在 ASAP 算法的第一轮迭代过程中，所有的任务都不基于开始时间设置为 1 的其他计算。在第二轮迭代过程中，从乘法得出的输入还是不可用的。在第三轮迭代过程中，减法在时刻 3 被调度。该过程直到最终的加法在时刻 7 被调度后方可终止

㊀　每一个整数都假定与同步自动机的时钟周期相对应。

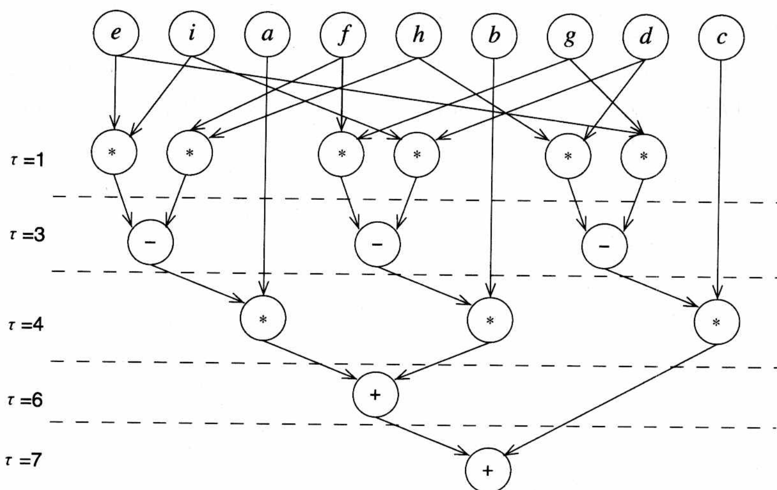



图 6.14　基于图 6.13 示例的 ASAP 调度

运行。

ASAP 调度也可以在现实中应用：它意味着所有的任务都要尽可能早开始运行，并且不考虑任何资源限制。

#### 6.2.3.3　ALAP 调度

ALAP 调度是第二个简单调度算法。对 ALAP 调度来说，所有的任务要尽可能晚运行。该算法工作方式如下：

**for** ($t$=0; 所有任务都被调度；$t$- -) {

  $s$={所有任务的运行都不需要其他任务的依赖关系};

  设置所有任务$t$的起始时间为$s$-其执行时间 + 1;

  }

 添加所有需要的起始时间的时间戳的总数

该算法从一个不基于其他任务依赖的任务开始运行，这些任务假定在时刻 0 时结束运行。这些任务的开始运行时间可以由其执行时间运算得出，并随着时间的变化向后循环迭代运行。每当到达一个任务最迟结束的时间点时，一个新的任务起始时间被计算得出并且开始任务的调度。当循环结束后，所有的时间都转为一个绝对时间，这样接下来的第一个任务又在时刻 1 开始运行。也可以认为 ALAP 调度是 ASAP 调度在“其他”图的结束开始运行的特例。

图 6.15 显示了基于图 6.13 的数据流图的调度结果。

对于 ALAP 调度，4 个“任务”在正确的时间单元后运行。

#### 6.2.3.4　LS 调度

LS 调度是一种有资源限制的调度技术。假设有一组 M 类型的资源，LS 调度在每个任务都只有在握有某种特殊类型资源时方可执行。表调度关心每个 $m \in M$ 类型

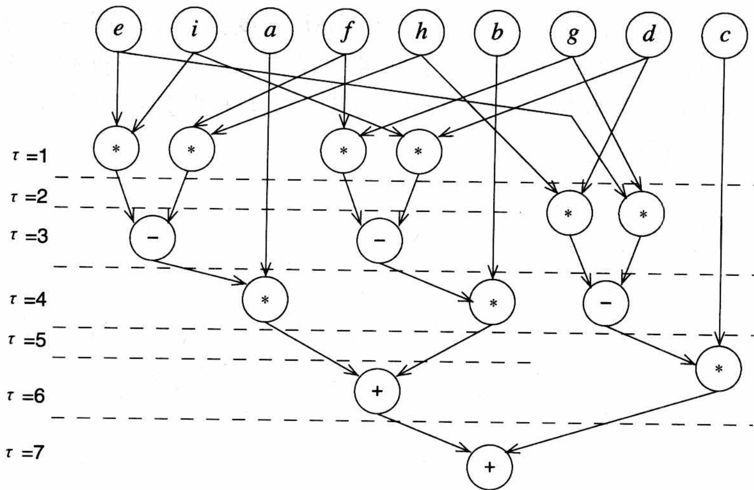



图 6.15 使用图 6.13 的 ALAP 调度

在上界 $B_m$ 的数量。

LS 调度需要获知一些具有某些优先级的、可用的、可以反映调度紧急性的特殊“任务” $v \in V$，$G=(V, E)$ 信息。使用如下指标度量调度紧急性 [Teich, 1997]：

1）后续节点的节点数量：该数量为在树中当前节点 $v$ 下的节点数量。

2）路径长度：节点 $v \in V$ 的路径长度为由 $v$ 开始到图 $G$ 末尾节点的路径长度。在图 6.16 中，加入了关于路径长度的相关信息。路径长度通常包含有与这些执行节点相关的执行时间的权值。在此假设执行时间是已知的。

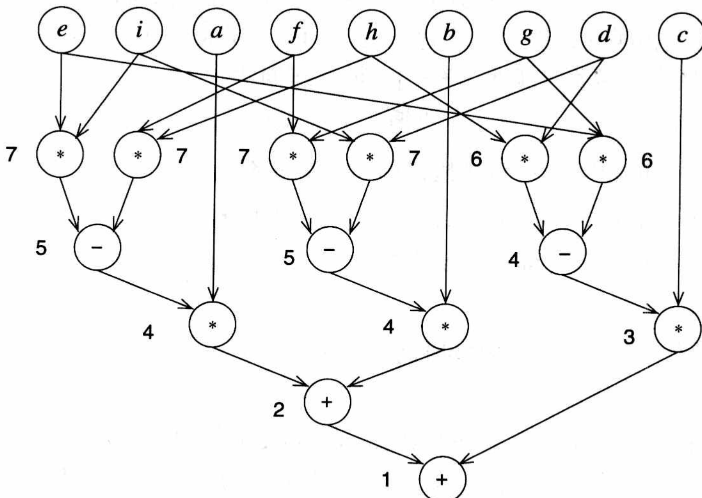



图 6.16 基于图 6.13 的路径长度

3）迁移率：迁移率定义为 ASAP 以及 ALAP 调度器的不同起始时间。图 6.17 展示了对应示例的迁移率。很明显，除了 4 个节点外，调度安排有非常紧迫的压力。这意味着所有的节点都拥有同样的优先级，并且通过迁移率只可以了解到所要调度任务的粗略顺序信息。

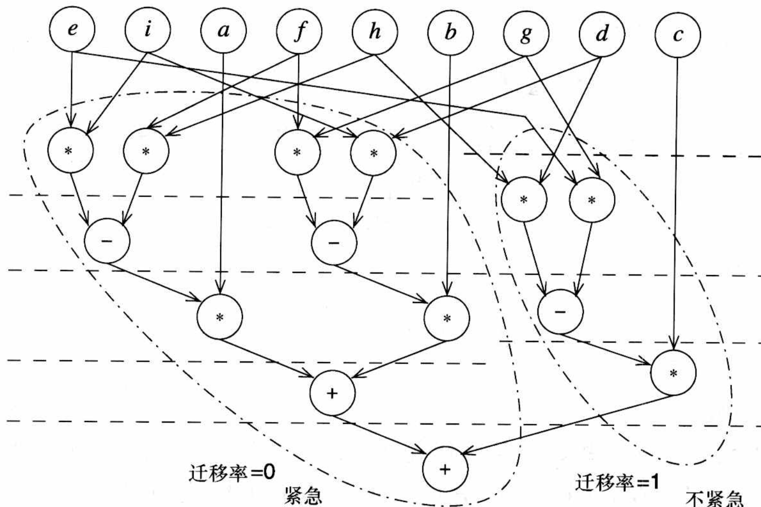



图 6.17 基于图 6.13 的迁移率示例

LS 调度需要获知欲调度的图 $G=(V, E)$ 的相关信息。对每个 $m$ 的上界 $B_m$ 而言，图中每个节点与对应的资源 $m \in M$ 的映射反映节点 $v \in V$ 的紧急性的优先级函数，以及每个节点 $v \in V$ 的执行时间。LS 调度接下来会尝试将每个具有最大优先级的节点与时间戳相匹配，以保证不违反资源约束［Teich，1997］：

**for** ($t$=0; 所有任务都被调度; $t$++) { //循环时间步

**for** ($m \in M$) { //循环资源类型

$C_{t,m}$ = 一组在$t$时刻还始终执行的$m$类型任务;

$A_{t,m}$ = 一组在$t$时刻准备开始执行的$m$类型任务;

计算 $S_t \subseteq A_{i,m}$ 中最高优先级的任务

$|S_t| + |C_{t,m}| \leq B_m$.

设置所有 $v \in S_t$ 的开始运行时间为 $t$: $\tau(v) = t$;

}}

图 6.18 显示了基于图 6.13 的示例的 LS 调度的结果。

在图 6.13 中，假定资源限定 $B_*=3$ 用于乘法与乘数，并且 $B^{+,-}=2$ 用于所有其他的“任务”。由于资源限制，3 个乘法在时刻 3 运行而非时刻 1。其他操作的资源限定没有任何影响。请牢记乘法需要两个时钟步长。

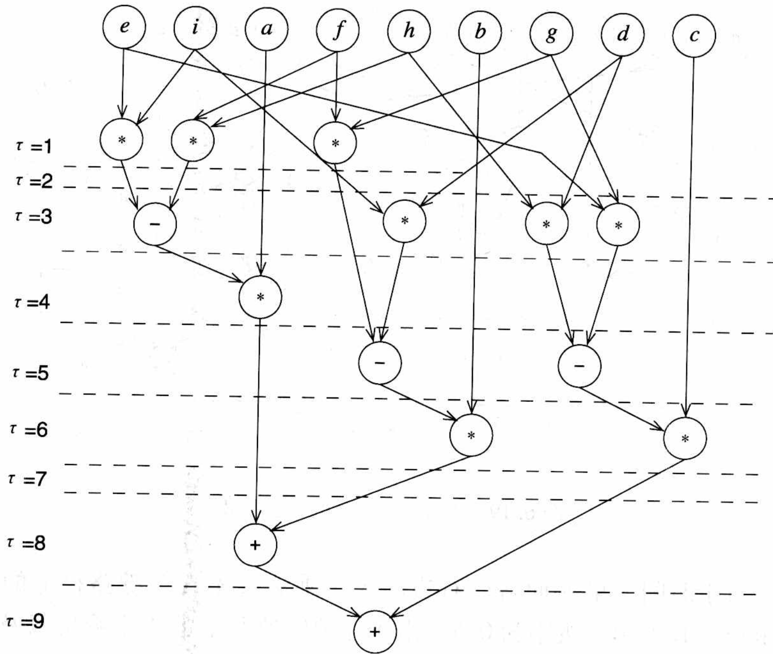



图 6.18　基于图 6.13 示例的 LS 调度的结果

#### 6.2.3.5　FDS

对于 FDS［Paulin and Knight，1987］，假设给定了时间限制，并且希望找到一个有资源约束的调度表，以最大限度地减小对资源的需求。FDS 单独考虑每种资源类型。

FDS 从一个能够反映某个特定操作 $v$ 在某一时刻 $t$ 的可调度的"概率"$P(v, t)$ 处开始运行。这个"概率"等于 1 除以 $R(v)$ 的大小。这里的 $R(v)$ 指的是一组操作可能开始的时间的集合：

$$P(v,t)=\begin{cases}\dfrac{1}{|R(v)|}, t\in R(v)\\ 0, \text{其他}\end{cases}$$

式中　$R(v)$ ——由 ASAP 调度器以及 ALAP 调度器分配的时间步长之间的时间间隔。

通过这个"概率"，可以计算出所谓的能够反映某个资源 $m$ 在控制步 $t$ 下的总体资源压力的"分布"。这个"分布"仅仅是对所有资源类型 $m$ 的操作的可能性总和：

$$D(t)=\sum_{v\in V}P(v,t)$$

图 6.19 显示了运行示例的分布。

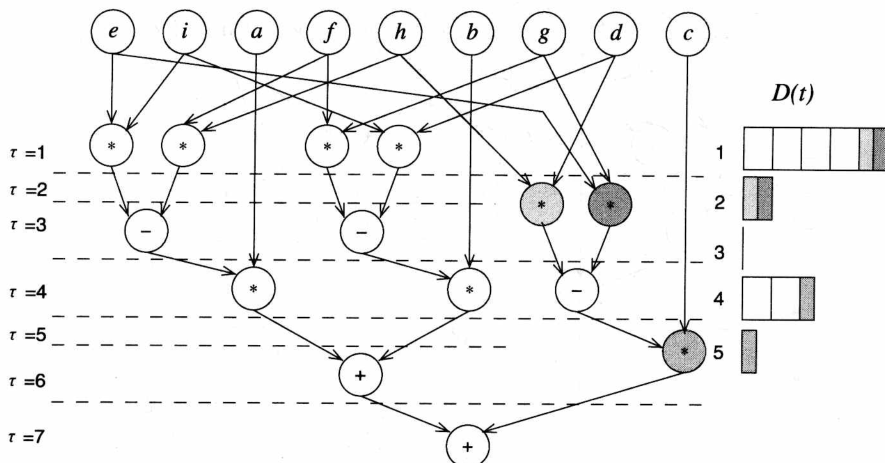



图6.19　基于图6.13的分配示例

例如，对于时间约束8而言，右边的3个乘法（不在关键路径上的）有两个可用时间步长，其可用的概率为0.5。由于在关键路径上的4个乘法以及两个概率为0.5的乘法，其分布 $D(1)$ 就是5。

接下来，FDS规定“力”，使得操作（或任务）从有高资源压力的时间步长上移动至较低资源压力的时间步长上。在这里的例子中，不在关键路径上的乘法的起始时间被向后移动，然而对于总的时间限制8来说，由于乘法假定为持续两个时间步长，这并不能降低所需乘数的数量。对于时间约束9来说，相比于ASAP调度的乘数，可以降低乘数的数量。有关FDS的更多细节可以参考［Paulin and Knight, 1987］的相关论文。

FDS有着若干种限制。例如，FDS依旧基于简单资源模型，每个任务只能被映射到一种资源上。

现在，有一种被称作HLS的调度技术用于从较高层次看待系统整体的调度行为。

HLS技术如下：

1）设计用于要考虑的“任务”之间的依赖关系；

2）设计用于“多处理器”的调度；

3）通常基于简化资源（处理器）模型（也就是“任务”和“处理器”之间的一一映射）；

4）通常使用穷举法，并且不保证最优性能；

5）速度通常比较快；

6）几乎从不使用周期性的全局信息等；

7）比ASAP、ALAP以及LS在控制上面的处理技术更加先进（循环等）。

### 6.2.4 没有优先级约束的周期调度

#### 6.2.4.1 注释

接下来，将考虑周期性任务。对于周期性调度，考虑有关非周期性调度的目标是没有意义的。例如，当讨论一个有关无限循环的任务时，最小化调度表的总长度就不是问题，所能做的就是设一个总能找到存在的调度表的算法，这促使人们去定义最优的周期调度表。

**定义**：对于周期性调度，如果在存在可用调度表的情况下，调度器总能找到存在的调度表，那么这个调度器就是**最优的**。

假设 $\{T_i\}$ 是一组任务。任务 $T_i$ 的每次执行被称作一次**作业**。每个任务的每次作业假设都是相同的，于是有（见图 6.20）：

1）$p_i$ 是任务 $T_i$ 的周期；

2）$c_i$ 是 $T_i$ 的执行时间；

3）$d_i$ 是**死线的间隔**，换言之，就是 $T_i$ 的一次作业从开始到必须结束的时间；

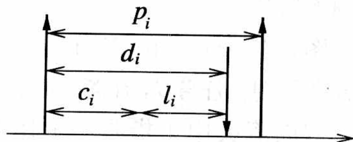



图 6.20 用于时间间隔的注释

4）$l_i$ 是任务的**闲置**时间，定义为

$$l_i = d_i - c_i \tag{6.4}$$

如果 $l_i = 0$，那么任务 $T_i$ 在变为可执行状态时就要立刻开始执行。

用 $\mu$ 表明一组 $n$ 个处理器的**利用率**，换言之就是用这些处理器累积的执行时间除以它们的周期：

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \tag{6.5}$$

假设 $m$ 个处理器的执行时间是相等的。很明显，式（6.6）代表了调度器存在的必要条件：

$$\mu \leqslant m \tag{6.6}$$

在系统的最开始，将在所有任务都是独立的场景限制下对具体的执行进行描述。

#### 6.2.4.2 单调速率调度算法

单调速率（Rate Monotonic，RM）调度［Liu and Layland，1973］可能是独立周期处理中，最知名的调度算法。RM 调度基于如下假设（ “**RM 假设**”）：

1）所有的任务都有一个周期性的硬死线；

2）所有的任务都是独立的；

3）对于所有任务来说，$d_i = p_i$；

4）$c_i$ 对于所有已知的任务来说是常量；

5）上下文切换所需的时间是微不足道的；

6）对于单处理器以及 $n$ 个任务来说，接下来的等式适用于累积利用率 $\mu$：

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leqslant n(2^{1/n} - 1) \tag{6.7}$$

图6.21显示了式（6.7）右侧的部分。

右侧等式中的$n$大约等于0.7：

$$\lim_{n\to\infty} n * (2^{1/n} - 1) = \log_e(2) = \ln(2)( = \sim 0.7) \tag{6.8}$$

根据单调速率调度，**任务的优先级是随着任务的周期单调递减的**。换句话说，具有较短周期的任务具有较高优先级，具有较长周期的任务的优先级较低。RM调度是**基于固定优先级的抢占式调度策略**。

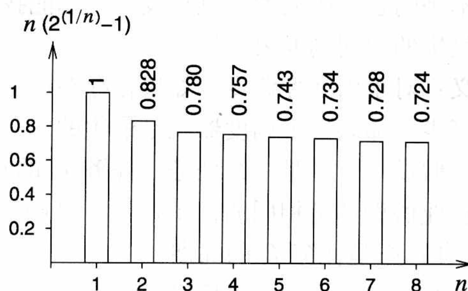



图6.21　式（6.7）的右侧部分

图6.22所示是由RM调度器生成的一个调度示例，任务$T_2$在若干时刻被抢占。

双向箭头用于指明一个作业的开始时间以及上一个作业的死线。任务1～3的周期分别为2、6以及6。执行时间分别为0.5、2以及1.75。任务1有着最短的周期，因此，其优先级最高。每当任务1转为运行态时，其作业会立即抢占当前正在运行的其他任务。任务2与任务3的周期相同，所以这两个任务不会抢占其他任务。

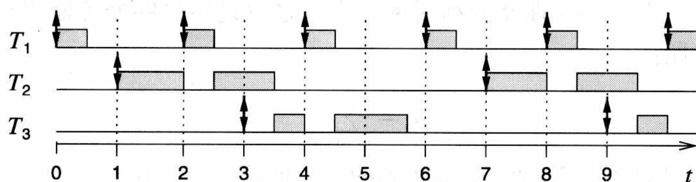



图6.22　由RM调度器生成的调度示例

式（6.7）要求处理器要保留一定的计算能力，以确保对所有的请求都能进行及时响应。保留一定的计算上限的原因是什么呢？主要原因就是RM调度器，由于该调度器使用静态优先级，这就导致一个拥有更高优先级，但是死线还较长的任务抢占一个已经非常接近其死线的任务。这样优先级较低的任务就会丢掉其死线。

图6.23显示了一个没有足够的可用时间保证RM调度器的调度性的示例。一个任务的周期为5，并且执行时间为3，第二个任务的周期为8，执行时间为3。

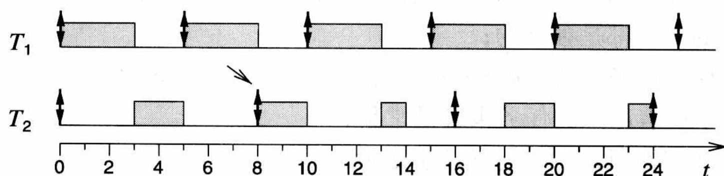



图6.23　RM调度器在时刻8没有满足死线要求

对这个特殊的例子，有$\mu=\frac{3}{5}+\frac{3}{8}=\frac{39}{40}$，$\mu$ 等于 0.975。$2*(2^{\frac{1}{2}}-1)$ 大约等于 0.828。因此，可调度性不能满足 RM 调度器的要求。此外，实际上在时刻 8，死线已经被错过。假定错过的指令会在下个周期被调度。

当处理器的利用率很低时，丢失死线的情况不会发生，并且当处理器的利用率很高时，就会发生丢失死线的情况，如图 6.23 所示。如果满足式（6.7）的条件，那么系统的利用率就会保证不会遇到图 6.23 所示的情况。式（6.7）是充分条件，这意味着有可能找到当不满足上述条件时的调度器。其他存在的充分条件参见 [Bini et al.，2001]。

RM 调度器具有如下重要的优点：

1）在多处理器系统上验证单调速率调度的最优性是可行的。

2）RM 调度器基于静态优先级。这就给了基于固定优先级的操作系统上使用 RM 调度的机会，例如 Windows NT 操作系统（见 Ramamritham [Ramamritham et al.，1998]，[Ramamritham，2002]）。

3）如果上述 6 种 RM 假设都能满足，那么所有的死线也会得到满足（见 Buttazzo [Buttazzo，2002] 相关著作）。

RM 调度也是以一定数量形式上的可调度性证明作为基础的。

处理器并不是总需要空闲时间以及备用产能的。显示 RM 调度器的最优性还是可能的，替换式（6.7），有

$$\mu \leq 1 \tag{6.9}$$

所有任务的周期应该是拥有更高优先级的任务的倍数。例如，该需求要求，如果某个任务在电视机中运行，那么其执行频率应该是 25Hz、50Hz 以及 100Hz。

式（6.7）以及式（6.9）提供了便捷的手段去检查可调度性的状态。

如果 RM 调度有若干不确定的状态，那么可以很方便地设计某些示例并证明。

**定义**：时刻 $t$ 被称作是任务 $T_j$ 的瞬时临界时间，如果任务在此时刻变为运行态，那么就说这个任务的响应时间是最大的。

**论点**：对于每一个任务 $T_j$，如果每个任务 $T_j$ 在同一时刻变为运行态并有最高优先级，那么响应时间就是最大的。

**证明**：对于一组周期任务 $T=\{T_1, \cdots, T_n\}$，有：$\forall i: p_i \leq p_i+1$。考虑到任务 $T_n$ 以及任务 $T_i$ 的高优先级（见图 6.24），随着任务的优先级变高，任务 $T_n$ 的响应时间也会增长。

如果 $T_n$ 以及 $T_i$ 可用的时间间隔降低，那么抢占的次数有可能增多（见图 6.25）。例如，图 6.24 的延迟为 $2c_i$，图 6.25 的延迟为 $3c_i$。

如果两个任务在同一时刻都变为可运行态，则高优先级的任务对低优先级任务的抢占会导致响应时间的增加。

与 $T_n$ 与 $T_i$ 相关的任务参数可以在其他成对出现的任务中重复使用。因此，如

果其他高优先级的任务同时被释放，$T_n$ 就可以在其临界时间到来的瞬间变为可执行态。

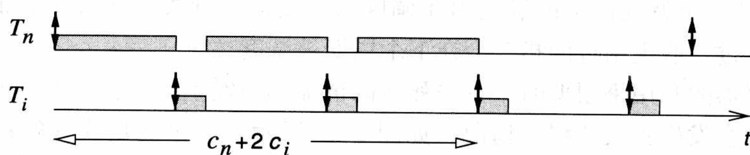



图 6.24　被高优先级任务 $T_i$ 延迟的任务 $T_n$

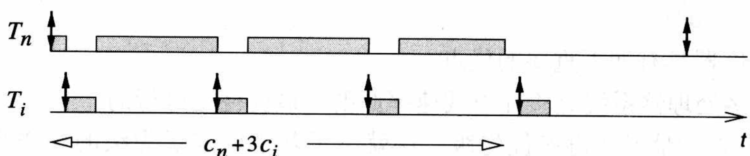



图 6.25　任务 $T_n$ 增加的延迟

因此，证明 RM 调度的最优性只需要考虑当拥有高优先级的任务同时释放时的场景。

#### 6.2.4.3　EDF 调度

EDF 调度也适用于周期性任务集。为此，可以考虑定义超周期的相关概念用于该调度集。

**定义：超周期**定义为单个任务周期的最小公倍数（least common multiple，lcm）。

例如，图 6.23 示例的超周期为 40。很明显，在一个超周期内就足以解决调度问题了。其他超周期也可以使用该调度方案。该调度方案基于非周期性的 EDF 调度法，适用于单超周期并解决相应的调度问题。由于没有额外的约束来保证必须达到最优性，这意味着在当 $\mu=1$ 的情况下，EDF 调度依旧是最优的。因此，当图 6.23的示例使用 EDF 调度（见图 6.26）时，不会发生死线丢失的情况。在时刻 5，EDF 的行为与 RM 调度的行为是不同的：由于其死线早于 $T_2$，所以不会发生抢占。

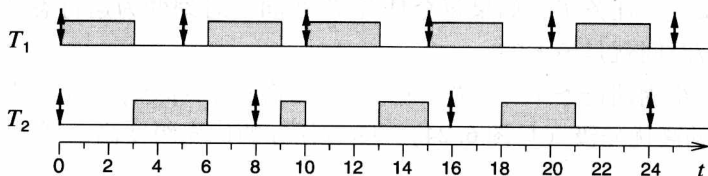



图 6.26　基于示例 6.23 使用 EDF 调度生成的调度表

由于 EDF 使用了动态优先级，所以该调度器不能与使用固定优先级的操作系统一起使用。然而，操作系统可以通过扩展，在应用层模拟 EDF 调度策略 [Diederichs et al.，2008]。

EDF 可以很容易就扩展为处理不同周期下的死线场景中。

### 6.2.5　有优先约束的周期调度

调度有相互依赖的任务比调度相对独立的任务更加困难。问题的关键在于对于给定的一组有依赖关系的任务以及给定的死线，调度器的存在与否取决于非完全多项式 [Garey and Johnson，1979]。为了减少调度工作，使用如下的不同策略：

1）添加额外的资源，使得调度变得更加简单。

2）将调度器分为静态以及动态部分。通过此方法，可以将更多的决策放在设计时，而较少的决策放在运行时。

很明显，也可以尝试使用基于 HLS 的技术用于周期进程。

### 6.2.6　零散事件

可以将零散事件组合为中断，当这些中断的优先级在系统中为最高优先级时，立即执行。然而，对于所有的任务来说，时间行为非常难以预测。因此，需要有一个特殊的零散服务任务用于执行定期以及检查准备好的零散任务。这样，零散任务本质上就可以转变为周期任务，从而提升这些任务在整个系统中的可预测性。

## 6.3　硬件/软件分割

### 6.3.1　简介

根据 6.1 节描述的问题，应用映射技术必须支持对异构处理器的映射。标准调度技术对这种映射的支持不够完善。然而，通过硬件/软件分割技术，可以对异构处理器映射做更好的支持。因此，在本节提出了一种针对该技术的示例。

通过硬件/软件分割，可以将任务图的节点同时映射至软件及硬件上。通过硬件/软件分割，可以决定哪个部分必须用硬件实现，哪个部分需要用软件实现。对嵌入式硬件/软件分割的标准流程设计流图如图 6.27 所示。从一个常用的表示规范开始，例如任务图表的方式以及平台信息的方式。

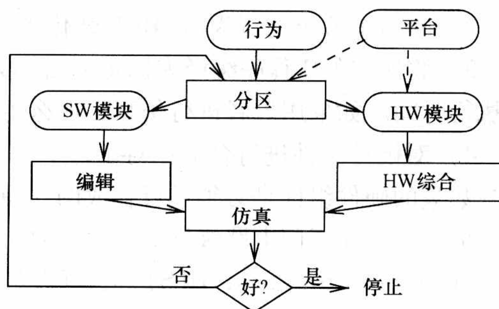



图 6.27　硬件/软件分割概貌

对于任务图的每个节点，需要知道所需工作量以及使用这些节点所能获得的收益。例如，执行时间必须可预计。预测通信所需的时间非常困难，然而如果两个任务需要非常高的通信带宽，那么将这

两个任务映射到相同的组件中就更加合适。许多情况下可以使用迭代的方法。对于解决分割问题的最初方法是产生、分析并改进。

一些分割的方法仅限于映射任务节点到特殊用途的硬件或运行于单处理器的软件上。这些分割可以用对分算法对任务图进行分割［Kuchcinski，2002］。

也有更多复杂的算法适用于对多处理器系统软/硬件的分割。接下来将描述这些算法如何通过标准优化技术以及**整数线性规划**进行（见附录A）分割，描述基于一个优化目的的设计工具COOL［Niemann，1998］的简单版本。

### 6.3.2 COOL

对于COOL，输入包括如下3个部分：

1）**目标技术**：COOL的这部分输入由可用硬件平台模块组成。COOL支持多处理器系统，但需要所有处理器的类型相同，因此COOL不包含自动或手动进行处理器选择。使用的处理器类型（以及相应的编译器）必须包含在COOL的这部分输入中。至于与某些硬件有关的应用程序，根据所有需要参数自动生成的硬件信息必须非常充分。特别是，有关这些技术的库的信息必须给定。

2）**设计约束**：第二个输入部分由例如需求效率、延迟、最大内存大小或最大程序制定硬件面积组成。

3）**行为**：输入的第三部分描述了对整体行为的需求。分层任务图用来描述该需求。可以使用图2.6所示的任务分层图进行相关分析。

COOL使用两种边界：通信边界以及时钟边界。通信边界包含交换信息数量的相关信息，时钟边界提供了时钟约束。COOL需要了解分层图中每个叶节点㊀的行为，COOL预期这种行为可以用VHDL指明㊁。

对于分割，COOL使用如下步骤：

1）将行为转换为**内部图模型**。

2）将每个节点的行为由**VHDL转化为C**。

3）将所有的**C程序编译**为选定的目标处理器类型，计算得到的程序大小，估算执行时间。如果用后者进行仿真，那么仿真的输入数据必须是可用的。

4）**对硬件组件进行合成**：对于每个叶节点，限定应用的硬件是合成的。因此相当数量的硬件组件是必须进行合成的，硬件的合成不应过于缓慢。用于商业级别的合成工具专注于门电路级别的合成，对于COOL来说过于缓慢。然而，高级别合成（High-Level Synisters，HLS）工具可以工作于寄存器级别（使用地址、寄存器以及多路转换器用于替换门电路）提供了充足的合成速度。此外，这些工具可以

---

㊀ 见2.4.2.1节相关定义。

㊁ 回顾已知C应当用于处理此类场景，但是在许多标准描述中，使用该工具相比于C而言更加简单。

提供足够准确的值用于时延以及所需的硅面积。在实际应用中，通常使用高级别合成工具 OSCAR［Landwehr and Marwedel，1997］。

5）**扁平的分层结构**：下一步是从分层流图中提取扁平任务图。由于没有对节点的合并或拆分，就需要设计者进行粒度维护。耗费以及性能信息可以从添加到节点中的编译以及硬件合成中获得。这实际是 COOL 的一个核心理念：**硬件/软件分割所需的信息可以预先计算，并且具有良好的计算精度**。在此信息上形成的最小化成本设计可以满足设计约束。

6）**生成和解决优化问题的数学模型**：COOL 使用整数线性规划（Integer Linear Programming，ILP）解决优化问题。商用的 ILP 程序用于找到决策变量的值，以最小化系统开支。解决方案就是从可用信息中获取开支函数，并进行优化。然而，该开支仅包括粗略的通信时间的近似值。任务图中任意两个节点的通信时间取决于这些节点所映射的处理器以及硬件。如果这两个节点映射到了同一个处理器上，那么通信只会在本地进行，因此速度可以非常快。如果节点映射到了不同的硬件模块，那么异地的通信速度有可能会比较慢。对所有的任务图的节点建立所有可能的映射模型会非常复杂，并且会被最初解决方案迭代的改进所取代。更多有关步骤的细节将会在下面进行描述。

7）**迭代的改进**：为了能够更好地对通信时间进行估算，可以将相邻的节点合并映射到相同的硬件模块中。该合并如图 6.28 所示。

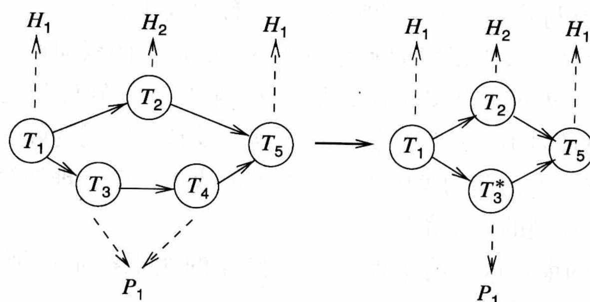


图 6.28　合并任务节点并映射至相同的硬件模块

假设任务 $T_1$、$T_2$ 以及 $T_5$ 映射至硬件模块 $H_1$ 以及 $H_2$，任务 $T_3$ 以及 $T_4$ 映射到处理器 $P_1$。这样，$T_3$ 与 $T_4$ 之间的通信就是局部通信，因此合并 $T_3$ 以及 $T_4$，并且假设这两个任务间通信不需要通信通道，通信时间的估算精准度也得以提高。结论图接下来可以用作数学优化时的新输入。接下来重复类似的步骤，直至所有的任务图节点都被合并。

8）**接口合成**：在分割后，需要创建用于连接处理器、特定应用硬件以及内存的粘合逻辑。

接下来将描述如何使用 0/1-ILP 模型进行分割建模（见附录 A）。接下来的索

引集用于描述ILP模型：

1）索引集 $V$ 表明了任务图节点，每个 $v \in V$ 相当于一个任务图节点。

2）索引集 $L$ 表明了任务图节点的类型。每个 $l \in L$ 都对应于一个任务图节点类型。例如，节点有可能描述平方根、离散余弦变换（Discrete Cosine Transform，DCT）或离散快速傅里叶变换（Discrete Fast fourier Transform，DFT）计算。每种计算都对应一种类型。

3）索引集 $M$ 指明了硬件组件的类型。每个 $m \in M$ 都与一个硬件组件类型相对应。例如，某些特殊的硬件组件是用于DCT或DFT。那么就有一个索引值与DCT硬件组件对应，另一个索引值与DFT组件对应。

4）对于每个硬件组件，可能会有很多个拷贝或者说“实例”。每个实例都定义为索引 $j \in J$。

5）索引集KP指明处理器。每个 $k \in \mathrm{KP}$ 表明了一个处理器（所有的处理器都是同种类型）。

模型需要如下的决策变量：

1）$X_{v,m}$：如果节点 $v$ 映射至硬件组件类型 $m \in M$，则变量值为1，否则为0。

2）$Y_{v,k}$：如果节点映射至处理器 $k \in \mathrm{KP}$，则变量值为1，否则为0。

3）$\mathrm{NY}_{l,k}$：如果至少有一个类型 $l$ 节点映射至处理器 $k \in \mathrm{KP}$，则变量值为1，否则为0。

4）任务图节点与其对应类型的映射 $V \rightarrow L$ 称之为类型。

就特殊情况而言，成本函数为所有硬件单元的成本的累加值：

$$C = 处理器开支 + 存储器开支 + 特定应用硬件开支$$

如果在系统设计时不包括处理器、存储器以及特定应用的硬件，那么就可以明显缩减系统总开支。但由于约束因素，这并不能成为一个合理的解决方案。接下来描述一些基于ILP模型的约束条件：

1）**操作分配约束**：这种约束保证每个操作的实现都是在硬件或软件中完成的。对应的约束可以用下式表示：

$$\forall v \in V: \sum_{m \in M} X_{v,m} + \sum_{k \in \mathrm{KP}} Y_{v,k} = 1 \tag{6.10}$$

在纯文本中，这意味着对于所有的任务图节点 $v$，必须保证 $v$ 或者在硬件中实现（对于某些 $m$ 来说，设置某个 $X_{v,m}$ 中的变量值为1）或者在软件中实现（对于某些 $k$，设置某个 $Y_{v,k}$ 的变量值为1）。

所有的变量都假设为非负整数：

$$X_{v,m} \in IN_0 \tag{6.11}$$

$$Y_{v,k} \in IN_0 \tag{6.12}$$

附加约束保证决策变量 $X_{v,m}$ 以及 $Y_{v,k}$ 用1作为上界，因此事实上是一个0/1值的变量：

$$\forall v \in V: \forall m \in M: X_{v,m} \leqslant 1 \tag{6.13}$$

$$\forall v \in V: \forall k \in \mathrm{KP}: Y_{v,k} \leqslant 1 \tag{6.14}$$

如果某功能类型节点 $l$ 映射至某处理器 $k$，则该处理器的指令存储器必须包含该功能的软件拷贝：

$$\forall l \in L, \forall v: \mathrm{Type}(v) = c_l, \forall k \in \mathrm{KP}: \mathrm{NY}_{l,k} \geqslant Y_{v,k} \tag{6.15}$$

在纯文本中，这意味着对于所有的任务图节点类型 $l$ 以及所有的该类型节点 $v$，必须保证如果 $v$ 映射至某处理器 $k$（有 $Y_{v,k}=1$），那么处理器 $k$ 必须提供与功能 $l$ 所对应的软件，并且对应的软件必须在该处理器上是已存在的（有 $\mathrm{NY}_{l,k}=1$）。

附加约束确保决策变量 $\mathrm{NY}_{l,k}$ 依然是一个 0/1 值变量：

$$\forall l \in L: \forall k \in \mathrm{KP}: \mathrm{NY}_{l,k} \leqslant 1 \tag{6.16}$$

2）**资源约束**：下一组约束确保“不会有太多”的节点在同一时间映射至同一硬件组件中。假设对于每个时钟周期，每一个硬件组件在同一时刻只能执行一种操作。不幸的是，这意味着分割算法不得不为执行任务图节点生成分割调度器。对于大多数相关问题的实例，调度器本身就是一个非完全多项式问题。

3）**优先约束**：该约束确保调用于执行操作的调度器与任务图中的优先约束一致。

4）**设计约束**：该约束用于限制某些硬件组件的成本，例如存储器、处理器或特定应用硬件的面积。

5）**时间约束**：如果时间约束出现在 COOL 的输入中，那么就可以转化为 ILP 约束。

6）还有某些不太重要的约束没有包含在上述列表中。

示例：接下来将会在图 6.29 中展示这些约束是如何从任务图中形成的（与图 2.6 中一致）。

假设有一个硬件组件库，包含组件类型为 $H_1$、$H_2$、$H_3$ 的 3 个组件，每个组件的成本单元分别为 20、25 和 30。此外，假设还有一个成本为 5 的处理器 $P$。另外，假设图 6.30 的表描述了任务在这些组件下的执行时间。

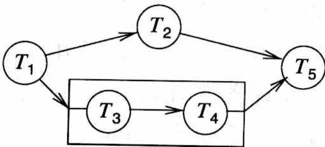



图 6.29　任务图

| $T$ | $H_1$ | $H_2$ | $H_3$ | $P$ |
|---|---|---|---|---|
| 1 | 20 | | | 100 |
| 2 | | 20 | | 100 |
| 3 | | | 12 | 10 |
| 4 | | | 12 | 10 |
| 5 | 20 | | | 100 |

图 6.30　任务 $T_1 \sim T_5$ 在组件上的执行时间

任务 $T_1 \sim T_5$ 只能在处理器或某一特定应用硬件单元上执行。很明显，这里假设处理器的成本更低，并且在执行任务 $T_1$、$T_2$ 和 $T_5$ 时更慢。

必须生成下列操作分配约束，假设使用最快的处理器（$P_1$）：

$X_{1,1}+Y_{1,1}=1$（任务1映射至 $H_1$ 或 $P_1$）

$X_{2,2}+Y_{2,1}=1$（任务2映射至 $H_2$ 或 $P_1$）

$X_{3,3}+Y_{3,1}=1$（任务3映射至 $H_3$ 或 $P_1$）

$X_{4,3}+Y_{4,1}=1$（任务4映射至 $H_3$ 或 $P_1$）

$X_{5,1}+Y_{5,1}=1$（任务5映射至 $H_1$ 或 $P_1$）

此外，假设任务 $T_1 \sim T_5$ 的类型分别为 $l=1$、2、3、3以及1，接下来需要额外的资源约束：

$$\mathrm{NY}_{1,1} \geqslant Y_{1,1} \tag{6.17}$$

$$\mathrm{NY}_{2,1} \geqslant Y_{2,1}$$

$$\mathrm{NY}_{3,1} \geqslant Y_{3,1}$$

$$\mathrm{NY}_{3,1} \geqslant Y_{4,1}$$

$$\mathrm{NY}_{1,1} \geqslant Y_{5,1} \tag{6.18}$$

式（6.17）表明：如果任务1映射至处理器，那么功能 $l=1$ 必须在该处理器上实现。如果任务5也映射至该处理器，相同的功能也必须在该处理器上实现。

这里并没有包括对时间的约束。很明显当处理器在执行某些任务和有低于100个时间单元的时间约束的特定功能硬件时，速度较慢。

成本函数为

$$C=20*\#(H_1)+25*\#(H_2)+30*\#(H_3)+5*\#(P)$$

这里#()表明了硬件组件的实际数量。如果将调度表也考虑在内，该数量可以通过引入的变量计算得出。对于有100个时间单元的时间约束，最低成本的设计包括组件 $H_1$、$H_2$ 以及 $P$。这意味着 $T_3$ 以及 $T_4$ 通过软件实现，而其余的通过硬件实现。

总之，由于对分区的组合以及调度问题的复杂性，只有非常小的组合问题可以在运行时解决。因此关键的问题是要将问题自我分割为调度问题及分割问题：最初的分割基于执行时间的估算以及分割后最终的调度。如果最终非常乐观地看待调度表，那么整个任务处理过程就必须使用更加严格的时间约束。实验表明，自我进行成本解决方案仅仅比最优化的成本有着1%或2%的成本增加。

自动分割可以用于对设计空间的分析。接下来将呈现一个由音乐制作室得出的结论，其中包括了混合、增益、回波、均衡器以及平衡单元。该示例使用了先前的目标技术，以证明分割的效果。目标硬件由SPARC处理器、外部存储器以及特定应用硬件组成，使用了1μASIC（早期的）库进行设计。系统允许的最大延时设置为22675ns，与用于CD的44.1KHz的采样率相对应。图6.31显示了不同的由改变延迟约束所得的设计要点。

单位λ是指与技术相关的长度单位。它本质上是芯片上两条金属导线中心最近距离的一半（也称作间距［ITRS Organization，2009］）。与左侧的设计点对应的为完全使用硬件实施的解决方案，右侧对应的是软件的解决方案。其他的设计点使

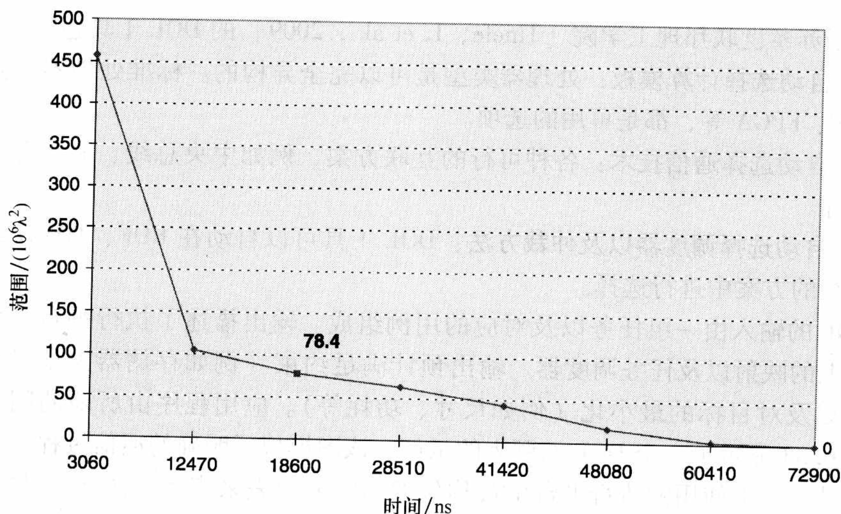



图 6.31　音频实验的设计空间

用了软、硬件混合的方案。78.4$\lambda^2$ 的面积是符合最便宜的要求的死线。

很显然，现今的技术已经发展到允许音频实验 100% 用软件实现，然而基于该设计方法的示例也可应用于更加苛刻的应用中，特别是高速多媒体领域，例如 MPEG-4。

## 6.4　映射至异构多处理器

目前（2010 年），异构多处理器的映射时钟是一个研究课题。欧洲卓越网络设计公司（the Artist Design European Network of Excellence）部门进行了应用映射至 MPSoC 的研讨会，对该领域的国内外现状进行了总揽。接下来讨论的内容不但基于第一次［Marwedel，2008a］以及第二次［Marwedel，2009a］研讨会，并对第一期研讨会的内容作了摘要的总揽［Marwedel，2009b］。对该映射的不同方法可以分为两个标准：映射工具可以假设用于固定的执行平台或者可以在映射时设计这样一个平台，该平台可能不包括自动并行化的源代码。图 6.32 包括了使用这两种标准的一些可用的映射工具的分类。

| 架构确定<br>自动并行化 | 固定架构 | 架构设计 |
|---|---|---|
| 从某个给定的模型开始 | HOPES, mapping to CELL proc., Q. Xu, T. Simunic | COOL, DOL, SystemCo-designer |
| 自动并行化 | Mnemee、O'Boyle和Franke<br>MAPS | Daedalus |

图 6.32　映射工具分类以及对应的开发者

源于苏黎世联邦理工学院［Thiele，L. et al.，2009］的 DOL 工具包括：

1）**自动选择计算模板**：处理器类型是可以完全异构的。标准处理器、微控制器、DSP、FPGA 等，都是可用的选项。

2）**自动选择通信技术**：各种可行的互联方案，例如中央总线、分层总线、环形总线等。

3）**自动选择调度器以及仲裁方法**：DOL 工具可以自动在 EDF、TDMA 以及基于优先级的方案中进行选择。

DOL 的输入由一组任务以及对应的用例组成。输出描述了执行平台、任务在处理器上的映射以及任务调度器。输出预计满足约束（例如存储器大小以及时间约束）以及对目标的最小化（例如尺寸、功耗等）。应用程序由所谓的问题表表示。图 6.33 显示了一个 DOL 问题表的示例，该图模拟了明确的通信过程。

此外，可能使用的执行平台由被称作架构图的图表来表示。图 6.34 展示了一个简单地用架构图表示的硬件平台。同样，通信过程被显示地建模。

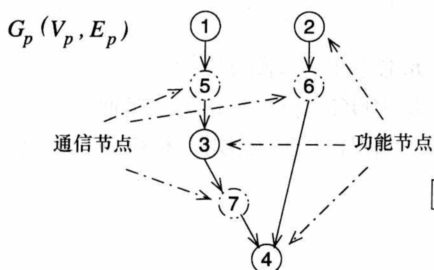



图 6.33　DOL 问题图

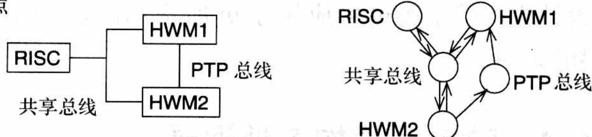



图 6.34　DOL 架构图

问题图以及架构图被合并为规范图。图 6.35 显示了 DOL 规范图。

该规范图包括了问题图以及架构图。两个子图的边界代表了可行的实现。总的来说，实现由一个三元组表示：

1）**分配 $\alpha$**：$\alpha$ 是架构图的一个子集，表明了硬件组件在特定设计下的分配（选定的）。

2）**绑定 $\beta$**：规范和体系结构所选择的子集的边界，表明了两者之间的关系。所选择的边界被称之为绑定。

3）**调度 $\tau$**：$\tau$ 分配了问题图中每个节点 $v$ 的开始时间。

示例：图 6.36 显示了如何将图 6.35 所示的规格说明书转化为一个实际的实现。

在 DOL 中，具体实现始于进化算法一起生成［Bäck and Schwefel，1993］，［Bäck et al.，1997］，［Coello et al.，2007］。通过这些算法，解决方案由独立的染色体字符串表示。使用进化算法，可以在现有的解决方案集合中生成新的解决方案的集合。集合的生成基于进化操作，例如突变、选择以及重组。在解决方案的新集合中进行选择基于适应值。进化算法能够解决复杂的优化问题，并且不像其他类型

的算法那样僵化。在染色体上寻找合适的编码方案并不是件容易的事。一方面，解码不应该使用太多的运行时间，另一方面，必须处理某些进化后的情况。除了一些精心设计的编码，这些进化有可能生成不可行解。

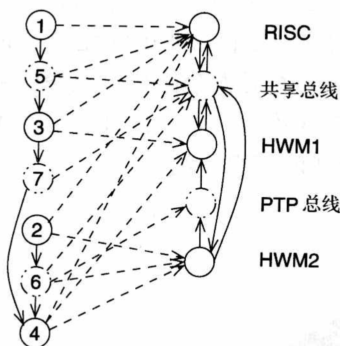



图 6.35　DOL 规范图

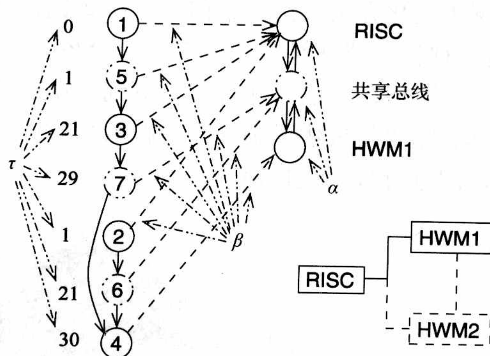



图 6.36　DOL 实现

在 DOL 中，染色体进行分配和绑定编码。为了对某一解决方案的适应值进行评估，分配以及绑定必须分别解码（见图 6.37）。

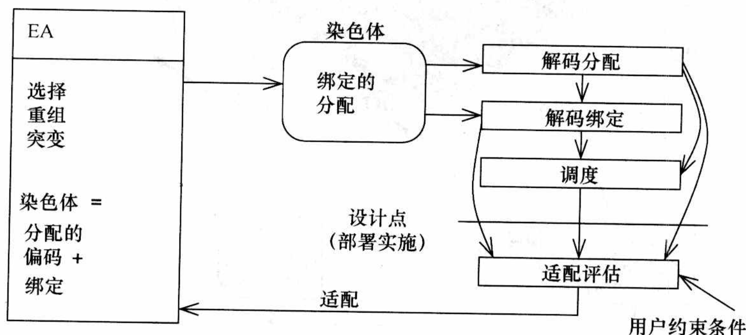



图 6.37　单个染色体的解码解决方案

在 DOL 中，染色体并没有对调度表进行编码，反之调度表是从分配及绑定中获得的。这种方法避免了对调度决策的进化算法进行重载。一旦对调度表进行计算，那么就可以对解决方案的适应度进行评估。

DOL 的总体架构如图 6.38 所示。

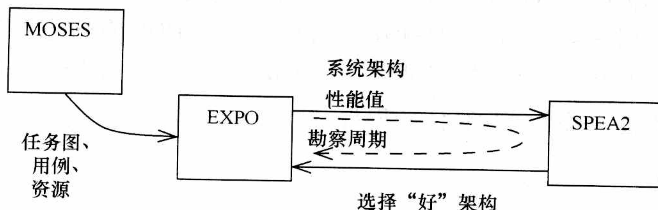



图 6.38　DOL 工具

在最开始，任务图、用例以及可用的资源都是已定义的。这些用例以及资源都可以通过一个被称作MOSES的特殊编辑器编辑。这些初始信息可以通过一个评估框架EXPO进行评估。可以通过EXPO计算出效率值，并发送至SPEA2，SPEA2是一个基于进化算法的优化架构。SPEA2选择好适合的候选架构，并将架构送回EXPO进行评估。评估结果会再次送入SPEA2中，进行下一轮的进化优化。这种在EXPO以及SPEA2的乒乓过程，直到发现优良的解决方案后方会停止。选择的解决方案基于帕累托最优原则。一组帕累托最优设计会返回给设计者，设计者会在不同目标之间进行分析及权衡。图6.39显示了基于帕累托的可视化结论。

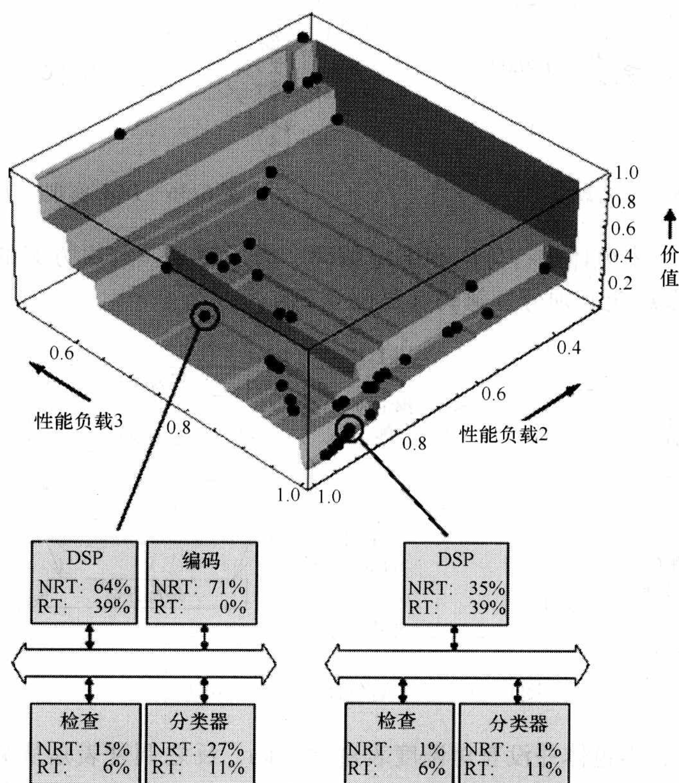



图6.39 使用帕累托最优原则解决相关设计问题的一个方案© ETHZ

SystemCodesigner［Keinert et al.，2009］的功能与DOL有些类似。然而，在规格的描述（可以用SystemC）以及优化的执行上，与DOL有所不同。应用程序的映射可以被仿真为ILP模型。通过使用ILP优化器，可以产生第一个解决方案。该解决方案可以通过切换至进化算法获得提升[㊀]。

㊀ 较新的版本使用可满足性（SAT）解决同样的目标。

Daedalus [Nikolov et al., 2008] 包含了自动并行处理。为了该目的，一系列的应用被映射至 Kahn 处理器网络中。通过 Kahn 处理器网络作为中间表示层，设计空间的探寻可以被继续执行。

其他方法从给定的任务图以及到固定架构的映射开始进行。例如，Ruggiero 可以将应用程序映射至 Cell 处理器 [Ruggiero and Benini, 2008]。通过使用 Ptolemy 工具提供的计算模型，HOPES 系统能够映射至多种处理器上 [Ha, 2007]。有些工具还将其他的目标也考虑在内，例如 Xu 考虑了可以对目标系统进行可靠性方面的优化 [Xu et al., 2009]。Simunic 在她的工作中包含了热分析，并且尝试在 MP-SoC 上避免出现热点 [Simunic-Rosing et al., 2007]。Popovici 等人将该工作进行了进一步发展 [Popovici et al., 2010]。通过使用 Simulink 以及 SystemC 语言，可以在若干级别进行仿真。

用于固定架构的自动并行化方法包括在爱丁堡大学使用的 [Franke and O'Boyle, 2005] Mnemee 工具集合 [Mnemee project, 2010]。MAPS 工具 [Ceng et al., 2008] 将 DSE 与自动并行化进行了结合。

## 6.5 思考题

1. 假设有一组 4 个任务。到达时间为 $A_i$，死线为 $d_i$，执行时间为 $c_i$，详情如下：

1）$T_1$: $A_1=10$，$d_1=18$，$c_1=4$

2）$T_2$: $A_2=0$，$d_2=28$，$c_2=12$

3）$T_3$: $A_3=6$，$d_3=17$，$c_3=3$

4）$T_4$: $A_4=3$，$d_4=13$，$c_4=6$

使用最早死线到达（Earliest Deadline First，EDF）以及最小余度（Least Laxity，LL）调度法进行任务调度，生成与该任务组对应的调度图！对于 LL 调度，指出所有任务在上下文切换时的最小余度，会有任务错过其死线吗？

2. 假设有一组 6 个任务 $T_1 \sim T_6$，其执行时间及死线如下：

1）$T_1$: $d_1=15$，$c_1=3$

2）$T_2$: $d_2=13$，$c_2=5$

3）$T_3$: $d_3=14$，$c_3=4$

4）$T_4$: $d_4=16$，$c_4=2$

5）$T_5$: $d_3=20$，$c_3=4$

6）$T_6$: $d_4=22$，$c_4=3$

任务的依赖性如图 6.40 所示。任务 $T_1$ 以及 $T_2$ 是立即可用的。

在使用 LDF 算法的情况下，生成基于该任务组的调度图。

3. 假设有一个包含两个任务的系统。任务 1 的周期为 5，执行时间为 2。任务

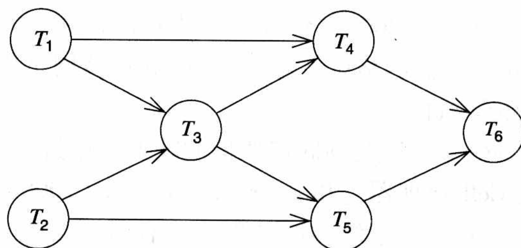



图6.40 任务依赖

2的周期为7执行时间为4。让死线与任务周期相同。假设使用RMS，在处理器使用率过高的情况下，这两个任务是否会错过其死线？计算其利用率，并在保证其调度性边界的情况下进行比较！假设这两个任务即使在错过其死线也能运行完毕的情况下，画出基于该调度系统的调度图表。

4. 考虑和思考题3. 同样的调度情况下，使用EDF是否会有任务错过其死线？如果没有，为什么？假设任务总能够运行完毕，画出基于此种调度算法的调度图。

5. 考虑一组任务。让 $V=\{v\}$ 为该组任务的索引。$L=\{l\}$ 为任务类型，并且类型：$V\to L$ 为任务到类型的映射。假设 $M=\{m\}$ 以及 $\mathrm{KP}=\{k\}$ 分别表示为硬件组件以及处理器。使用COOL描述下列硬件/软件模型分割：

1）需要使用哪些决策变量？

2）任务类型为 $l$ 的变量模型是否被映射到了处理器 $k$ 上。

3）目标函数应为何种运行方式？

4）哪些等式用来确保每个任务能够在硬件及软件上实施。

5）如果说该类型的任务可以运行在该处理器上，哪些等式用来确保该任务映射到该处理器中？

# 第7章　优　　化

为了让嵌入式系统尽可能满足性能需求，大量的性能优化方法被提出并不断发展。本书所能提及的只是这些算法的一个小子集。在本章，将提出一些可选择使用的优化方法。在设计流图中，如第6章以及图7.1所示，这些优化方法被当作应用程序到最终系统映射的补充工具。

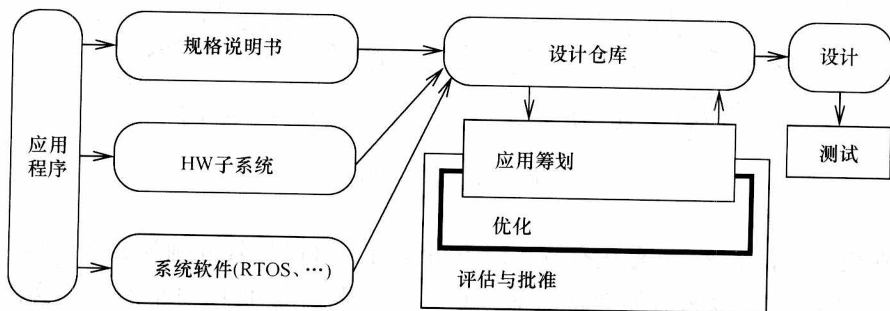



图7.1　本章所处的上下文环境

## 7.1　任务级并发性管理

如在2.2节所提及的，任务图的粒度是其最重要的属性。即使对于分层的任务图而言，改变其节点的粒度也是有必要的。对于旨在最大效率提升性能的目的，不一定要将系统规格说明书中的需求分割至任务或处理器中。当然，在规格阶段，清晰的结构划分以及干练的软件模型比更多的关心系统的实现更有意义。例如，一个清晰的结构划分包括了对使用的抽象数据类型的清晰划分。此外，有可能在系统需求中设计流水线，将若干任务在一条流水线中实现，以减少上下文切换时的开销。因此，在系统实现时将任务与设计书进行一一对应是不必要的，这意味着将任务进行重组是明智的。将任务重组，进行合并或分割是确实可行的。

每当任务 $T_i$ 是任务 $T_j$ 的直接前驱，并且 $T_j$ 没有其他任何直接前驱任务时（见图7.2，$T_i = T_3$ 并且 $T_j = T_4$）就可以进行任务图的合并。如果这些任务节点是通过软件实现的，通常来说，这种转变可以降低上下文切换的开销，并且为系统优化提供更多的潜能。

另一方面，进行任务分割有如下益处：

当任务在等待某些输入时，可以始终持有某些资源（例如大量的内存）。为了

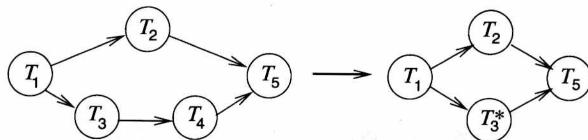



图7.2　任务的合并

最大限度利用这些资源，在这些资源实际需要之前，可以有限制地使用这些资源。在图7.3中，假设任务 $T_2$ 代码的某处等待某种输入。

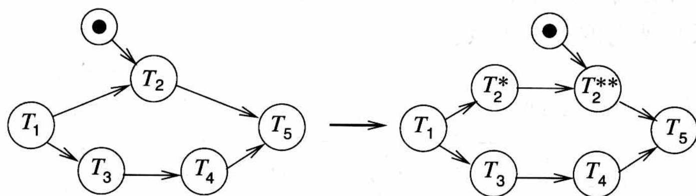



图7.3　任务分割

在最初，任务 $T_2$ 只有在有输入时方可运行。可以将其分割为两个节点 $T_2^*$ 以及 $T_2^{**}$，实际需要输入方能运行的部分是 $T_2^{**}$。现在 $T_2^*$ 可以更早运行，这样可以让调度更加灵活。这种分割可以提高资源的利用性，并且可以保证任务的运行不超过其死线时间。这种分割还会对数据存储所需的内存产生影响，因为 $T_2^*$ 可以在结束运行前不久释放某些内存，并且 $T_2^{**}$ 等待输入时可以将这些内存分配给其他任务使用。

有可能导致争论的地方是任务在等待输入之前就应当将所持有的资源，例如大量的内存释放。然而，由于系统规格书的可读性，在设计阶段的早期难以协调此类实现问题。

对于复杂的系统规范的转化可以通过 Cortadella 等人［Cortadella et al.，2000］所描述的基于 Petri 网的技术实现。这种技术由使用 FlowC 的语言所描述的一组任务开始进行。FlowC 语言对 C 语进行了进程头以及针对于进程间通信规范，例如 READ 以及 WRITE 函数调用的扩展。图7.4显示了使用 FlowC 的系统规范的输入。

示例中使用输入端口 IN 以及 COEF，输出端口为 OUT。点对点的进程间通信通过一个单向的缓冲通道 DATA 实现。任务 GetData 从外部读取数据并且将其发送至通道 DATA。每次有 $N$ 个样本被发送，其平均值也通过同一个通道发送。任务 Filter 从通道中读取 $N$ 的值（并忽略其内容），并读取其平均值，然后将平均值乘以 $c$（$c$ 可以通过 COEF 端口读取），最终将所得出的结果写至 OUT 端口中。READ 以及 WRITE 调用的第三个参数是要读和写的数据的数量。READ 调用是阻塞调用，如果通道上的数据数量超过了预先定义的阈值，WRITE 调用也会被阻塞。

SELECT 的语法结构与 ADA 是相同的：知道某个输入到达端口时，被阻塞的

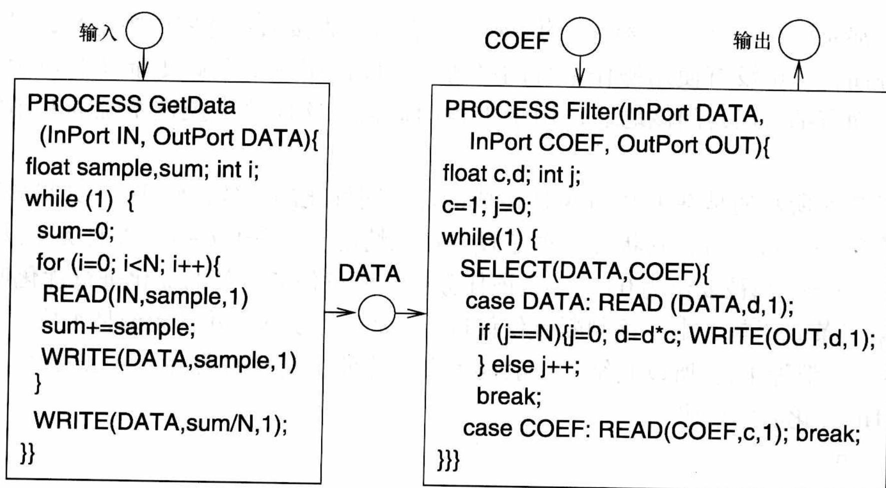



图 7.4　系统规格说明

任务才可执行。这个示例符合所有在图 7.3 中所提及的任务分割的标准，这两个任务将会在等待输入的同时占用资源。可以通过对这些任务的重组提升效率，然而仅仅对图 7.3 进行简单的分割是不够的。Cortadella 等人提出了一个更全面的技术，通过使用该技术，FlowC 程序首先会被转化成（扩展）Petri 网。每个任务的 Petri 网会被合并至一个 Petri 网中。通过 Petri 网，就可以生成一个新的任务。图 7.5 显示了一个可用的新任务结构。

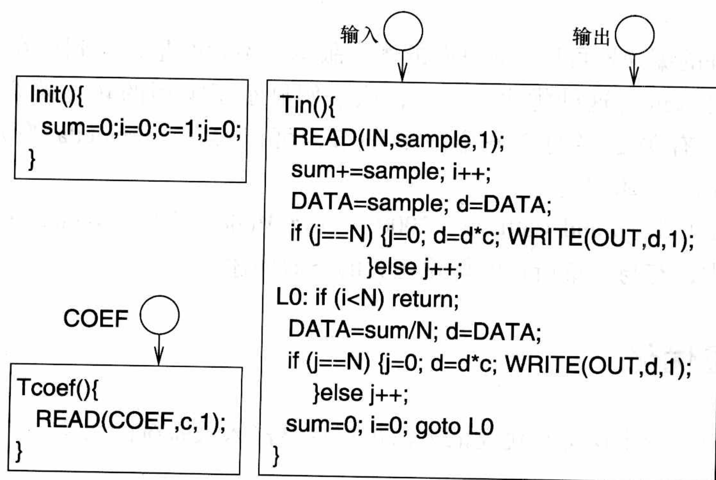



图 7.5　生成软件任务

在该新任务结构中，有一个任务会负责所有的初始化工作：此外，还会有一个任务用于所有的输入端口。该实现将会提升每个端口收到新输入后的中断效率。每

个端口都应该有一个属于该端口的中断，任务可以直接通过这些中断运行，对于这些任务而言，也没有使用操作系统的必要。可以使用一个全局变量（假设对于所有任务而言都可以访问该地址）用于进程间通信，这样整个操作系统的开销将会非常低。

图7.5所示的是基于Petri网所生成的任务间优化的任务代码Tin。如果该测试用例在第一次执行时，其状态经常是假（在此情况下$j$等于$i-1$，并且无论$i$是否等于$N$，$i$与$j$都被复位为0），那么该任务再次通过任务内部的优化进行优化调整。对于第三条if语句，由于该判断点仅仅进行$i$是否等于$N$，并且无论是否执行至L0的标号，$i$都等于$j$，所以其结果一般都为真。此外，变量的数量也可以减少。下面为对Tin优化后的代码：

```
Tin () {
  READ (IN, sample, 1);
  sum += sample;  i++;
  DATA = sample;  d = DATA;
L0:  if (i < N) return;
  DATA = sum/N;  d = DATA;
  d = d*c;  WRITE(OUT,d,1);
  sum = 0;  i = 0;
  return;
}
```

设计良好的编译器可以生成Tin的优化版本。不幸的是，直到现在也几乎没有哪个编译器可以进行这种优化。然而，该示例显示了类型的转化需要生成“好”的任务结构。有关更多的任务生成的细节，可以参考Cortadella等人所著书籍[Cortadella et al.，2000]。

在Thoen [Thoen and Catthoor，2000] 以及Meijer等人 [Meijer et al.，2010] 的相关著作中，有与本章所述的优化相关的类似描述。

## 7.2 上层优化

有许多对系统上层进行优化的方法可以提升嵌入式系统的效率。

### 7.2.1 浮点至定点转换

浮点至定点转换是一种常用技术，这种转换的原因基于许多信号处理标准（例如MPEG-2或MPEG-4）指定了在C程序中使用浮点数据类型，这使得程序开发者尝试找出基于这些标准更有效率的实现方法。

在许多信号处理程序中使用定点数替换浮点数是可行的，所能获得的收益也有可能是巨大的。例如，对于 MPEG-2 视频压缩算法［Hüls，2002］而言，使用该方法可以降低 75% 的循环次数以及 76% 的系统功耗，然而效率的提升会导致某些精度降低。更准确地说，开发者需要在系统开销以及算法质量中进行权衡［例如，对信噪比（SNR）的评估］。对于小字长系统，算法质量有可能影响更甚。因此，尽管可以使用定点数类型替换浮点数类型，但是精确度的丧失需要提前进行分析。这种数据的替换最初使用人工手动替换，然而这种替换方式有可能非常单调并容易导致错误的产生。

因此，有相应的研究者试图研发出某种可以进行自动替换的工具，FRIDGE［Willems et al.，1997］，［Keding et al.，1998］是其中的一种。FRIDGE 是 Synopsys CoCentric 工具包［Synopsys，2010］的一个商用组件。

在 FRIDGE 中，设计过程从一个包含浮点数的 C 算法开始进行。该算法会被转化成用 Fixed-C 描述的算法。Fixed-C 使用 C++ 的某些特性对 C 进行了两种定点数据类型的扩展。定点数据类型的声明与其他变量的声明类似。下面的声明分别进行了变量、指针以及数组的定点数据类型定义：

fixed a, *b,c[8]

有参数的顶点数据类型可以到数据使用时再进行存储空间分配（但不是必须如此）：

a = fixed(5,4,s,wt, *b)

上述语句分配给 a 5bit 字长，小数部分字长为 4bit 分配至 s 中，（w）用于数据溢出处理，（t）则用于数据舍入。这些参数在一次作业中被读入，并由该作业进行变量分配。数据类型 Fixed 与固定数据类型中的“固定”概念相仿，但是在做参数的一致性检查与分配时，有些微区别。对于变量的每次分配，其参数（包括字长）都有可能不同。在使用程序进行仿真之前，这些参数都应该添加到原始的 C 程序中。仿真程序会为这些参数提供若干变量的值。基于这些信息，FRIDGE 会将所有的参数信息添加到所有的作业中。FRIDGE 有时根据上下文的情况推算参数信息，例如最大值有可能是所有参数的总和，这些参数信息可以基于仿真器或者是基于最坏情况下的推断。在基于仿真的情况下，FRIDGE 无需进行基于正式分析的最坏情况的数据假设。接下来的 C++ 程序会对降低的效率进行仿真检测。Synopsys 版本的 FRIDGE 使用 SystemC 顶点数据类型表示生成的数据类型信息，因此 SystemC 可以用于进行定点数据类型的仿真。

Shi 和 Brodersen［Shi and Brodersen，2003］，此外还有 Menard 等人［Menard and Sentieys，2002］提出了一种在额外的噪声以及字长之间进行权衡的分析方法。

### 7.2.2 简单循环转换

有很多种循环转换的方法可以用于系统架构设计，下面为一些进行循环转换的

标准：

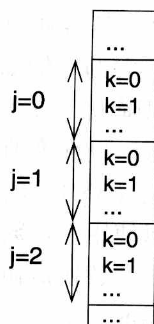


图7.6 C语言中二维数组P［j］［j］中的内存布局

1）**循环置换**：考虑一个二维数组。在C标准中［Kernighan and Ritchie，1988］，二维数组在内存中的分配如图7.6所示。相邻索引值的第二个索引被映射至了一块连续的存储空间上。这种分配方式被称作行主序［Muchnick，1997］。但要注意，在FORTRAN中，数组的布局与C语言是不同的：相邻值的第一个索引被映射到连续的存储空间（列主序）。有些描述FROTRAN优化的相关文章在此处有可能导致读者发生某些混淆。

对于行主序，最后的索引与最内部的循环相对应的循环组织方法通常来说对系统效率提升是有帮助的。下面的示例显示了一个相应的循环置换方法：

```
for (k=0; k<=m; k++)            for (j=0; j<=n; j++)
  for (j=0; j<=n; j++)     ⇒      for (k=0; k<=m; k++)
    p[j][k] = ...                   p[j][k] = ...
```

这种排列有可能会对高速缓存中的数组元素重用产生积极影响，因此接下来迭代的循环体会访问相邻的存储区域。高速缓存通常会对这些相邻的区域进行管理，以大大增强其访问效率。

2）**循环融合、循环裂变**：在某些情况下，需要将两个独立的循环进行合并，并且在某些情况下需要将单独的循环进行分割，示例如下：

```
for (j=0; j<=n; j++)            for (j=0; j<=n; j++)
  p[j]= ... ;                     {p[j]= ... ;
for (j=0; j<=n; j++)     ⇔        p[j]= p[j] + ...}
  p[j]= p[j] + ...
```

如果目标处理器提供了可以用于较小循环的零开销循环指令，那么使用左侧的版本可能会更有效率。右侧的代码版本有可能会提高缓存的使用率（由于数组p的访问存在局部性），并且会增加循环体的并行计算能力。与其他转换相比，上述的转换只能说对各个场景各有优势，很难界定哪种转换方式更加合理。

3）**循环展开**：循环展开是在创建若干循环体的实例时的标准转换方法，进行一次循环展开的示例如下：

```
for (j=0; j<=n; j++)            for (j=0; j<=n; j+=2)
  p[j]= ... ;              ⇒      {p[j]= ... ;
                                   p[j+1]= ...}
```

循环的复制数被称作展开系数，展开会降低循环的开销（与原始循环体相比，其分支更少）并提升系统运行速度。在某些极端情况下，可以将循环完全展开，

以此完全移除控制以及分支管理的开销。有时可以对循环进行若干次的转换变换，这样展开的程序与未展开或展开一次的相比会更有优势。然而展开会导致代码长度的增加，展开通常被限制为循环常数的迭代次数。

### 7.2.3　循环分块

处理器速度的提升是优于存储器速度的提升的。由于较小容量的内存与大容量内存相比有着速度优势，使用分级存储结构可能会对系统效率更有帮助。这些“小”容量存储器可能会包含缓存等存储器。使用这些存储器的原因是基于某些信息的重用因素，如果不考虑重用，那么分层结构的存储体系就毫无意义。

重用的效果可以通过分析下述示例证明，现在考虑一个 $N*N$ 的矩阵乘法 [Lam et al.，1991]：

```
for (i=1; i<=N; i++)
    for(k=1; k<=N; k++){
     r=X[i,k]; /* 分配至寄存器 */
     for (j=1; j<=N; j++)
      Z[i,j] += r* Y[k,j]
    }
```

下面考虑这段代码的访问方式。X [i，k] 在最内部的循环中被迭代使用。编译器会将该值放入寄存器中，并且在最内部的每次循环执行时进行重用。假设数组中的元素以行主序进行内存分配（与标准 C 一致）。这意味着相邻的一行为索引的数组元素存储在相邻的内存空间中。因此，在最内部的循环迭代中可以获取相邻的 Z 与 Y 的元素。如果存储系统使用预取技术，那么该技术对效率的提升非常有帮助（每当有一个机器字的数据被载入到缓存中，将相邻的下一个字也载入到缓存中）。图 7.7 显示了基于该代码的存储器访问方式。

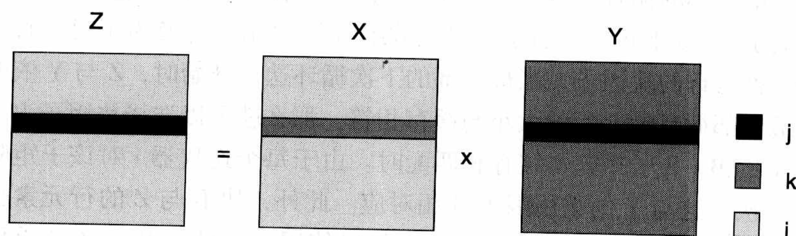



图 7.7　矩阵乘法的访问方式

对于最内部循环的一次迭代，数组 Z 以及 Y 被访问（并被载入至缓存中）。缓存中使用过的相同数据是否在下次的迭代循环中出现，取决于缓存的大小。在最坏的情况下（假设 N 的尺寸较大，但高速缓存容量较小），在最内部的循环每次执行

时，相关信息必须重新载入至缓存，此时缓存中的信息没有进行重用。可以在高速缓存中进行引用的存储空间的大小分别为 $2N^3$（参考 Z）、$N^3$（参考 Y）以及 $N^2$（参考 X）。

通过对访问局部性的深入研究以及科学计算方面的研究工作使得**循环分块**与阻塞技术得到了长足发展［Xue，2000］。下面为一个基于分块循环的算法示例：

```
for (kk=1; kk<= N; kk+=B)
  for (jj=1; jj<= N; jj+=B)
   for (i=1; i<= N; i++)
    for (k=kk; k<= min(kk+B-1,N); k++){
     r=X[i][k]; /* 分配至寄存器 */
     for (j=jj; j<= min(jj+B-1, N); j++)
      Z[i][j] += r* Y[k][j]
    }
```

图 7.8 显示了相应的访问方式。

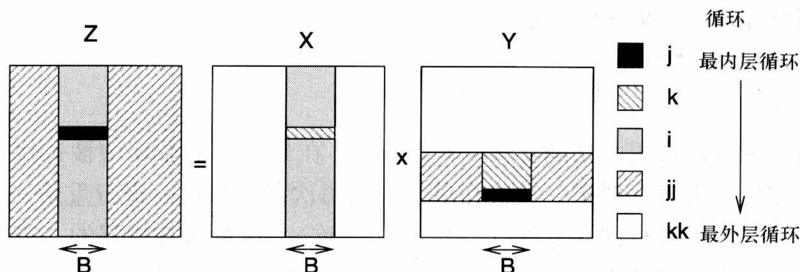



图 7.8　基于分块的矩阵乘法访问方式

为了让最内部的循环减少访存次数，故对最内层循环使用进行若干显示（图中用黑块显示）。如上述代码所示，对于数组 X 的引用被替换为了对 r 的引用。如果选择了某种可行的块因子，当最内部的下次循环迭代开始时，Z 与 Y 依旧在缓存中。如果最内部循环的元素的大小与缓存相符，那么就可以选择块因子 B。特别是当有一个 Y 的 B＊B 子矩阵与缓存相匹配时。由于每个迭代器 i 对该子矩阵的访问次数都为 B 次，这与 Y 的重用因子 B 相对应。此外，块 B 与 Z 的行元素也应与缓存是相符的。这些数据可以在进行 k 次迭代时使用，这导致 B 与 Z 的重用因子是一致的。这会至少 $2N^3/B$ 次对内存的总引用。实际上，其重用因子可能要低于 B。对重用因子的优化是一个综合的研究领域，最开始相关研究专注于对性能的提升。Lam［Lam et al.，1991］发表了关于因子为 3 与 4.3 的矩阵乘法性能研究分析的相关报告。接下来的研究方向是处理器与存储器之间越来越大的速度差距的性能

研究。分块也可以降低整个存储系统的功耗［Chung et al.，2001］。

## 7.2.4 循环分割

接下来将继续讨论用于程序编译之前的另一个优化方法，即循环分割。这个优化方法也可以添加到编译器中进行。

许多图像处理算法需要进行某些滤波处理。滤波时需要对某些特定像素以及邻近像素进行图像滤波。对这些像素进行的计算也比较普通，然而如果要处理的某些像素与图像的边界相邻，有可能没有相邻像素存在，并且计算方式必须进行修改。在某种滤波算法中，这些修改有可能在算法的最内部循环中被执行。但是更有效率的滤波算法会将处理普通情况下的循环体与处理某些异常情况的循环体分割。图7.9显示了相关的转换。

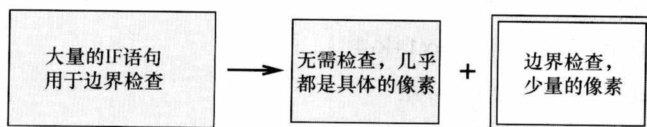



图7.9 将图像处理过程分割为常规与特殊情况

手动进行循环分割是一项非常有难度的工作，并且容易出错。Falk 等人发布了即使大尺寸图片也可对其进行自动处理的算法［Falk and Marwedel，2003］。该算法基于对循环数组访问的精密分析，通常使用遗传算法来生成最优的解决方案。下述代码显示了基于 MPEG-4 的动态估值的循环算法：

```
for (z=0; z<20; z++)
  for (x=0; x<36; x++) {x1=4*x;
   for (y=0; y<49; y++) {y1=4*y;
    for (k=0; k<9; k++) {x2=x1+k-4;
     for (l=0; l<9; ) {y2=y1+l-4;
      for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
       for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
        if (x3<0 || 35<x3||y3<0||48<y3)
         then_block_1; else else_block_1;
        if (x4<0|| 35<x4||y4<0||48<y4)
         then_block_2; else else_block_2;
}}}}}}
```

使用 Falk 的算法，该嵌套循环可以转化为如下形式：

```
for (z=0; z<20; z++)
  for (x=0; x<36; x++) {x1=4*x;
   for (y=0; y<49; y++)
    if (x>=10||y>=14)
     for (; y<49; y++)
      for (k=0; k<9; k++)
       for (l=0; l<9;l++ )
        for (i=0; i<4; i++)
         for (j=0; j<4;j++) {
          then_block_1; then_block_2}
   else {y1=4*y;
     for (k=0; k<9; k++) {x2=x1+k-4;
      for (l=0; l<9; ) {y2=y1+l-4;
       for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
        for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
         if (0 || 35<x3 ||0|| 48<y3)
          then_block_1; else else_block_1;
         if (x4<0|| 35<x4||y4<0||48<y4)
          then_block_2; else else_block_2;
}}}}}}
```

为了替换复杂的内部循环，现在将第三个 for 循环语句后的 if 语句进行分割。所有的常规图形处理都放入了后续的 then 语句中，Else 语句处理剩下相对较少的情况。

图 7.10 显示了对于不同的处理器以及应用来说，进行嵌套循环分割后可以节

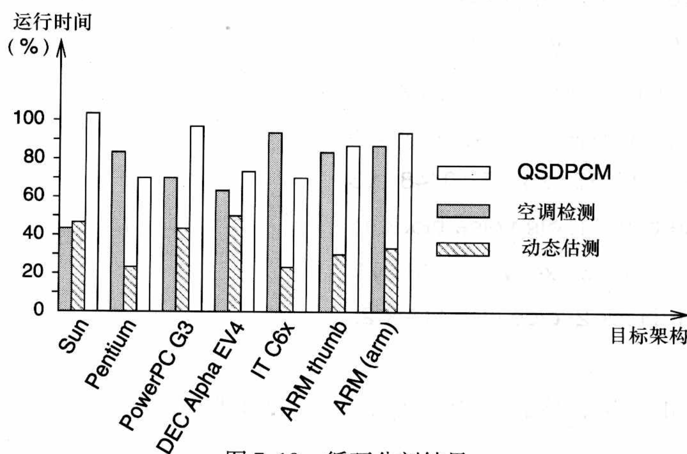



图 7.10　循环分割结果

省的循环周期的数量。

通过对算法的动态估算，使用循环分割算法后可以将循环次数最多降低75%（其原始循环次数的25%）。显然，对于循环嵌套类程序而言，其算法的优化空间巨大，而这种巨大的优化可能是无法被忽略的。

### 7.2.5 数组折叠

对于某些嵌入式应用，尤其是多媒体领域的应用，其代码内部包含着大量的数组。由于嵌入式系统的存储空间有限，这就需要找出一种能够降低数组所需存储空间的方法。图7.11显示了某函数在不同时间如何使用5个数组的地址。在任何的特定时间，只会使用数组元素的某一子集。所需元素的最大数量称为**地址引用窗**[De Greef et al.，1997b]。在图7.11中，地址引用窗由双向箭头表示。

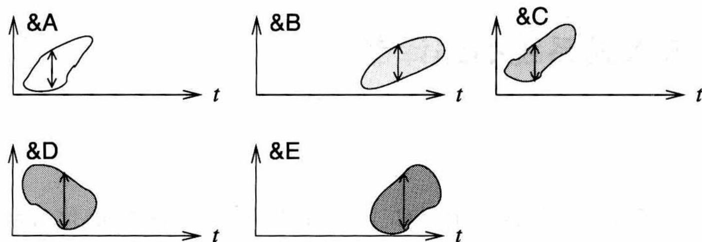



图7.11 数组的引用方式

经典的数组内存分配如图7.12左图所示。在进程执行的所有时间内，每个数组所需的最大空间都会被分配（假设考虑的是全局数组）。

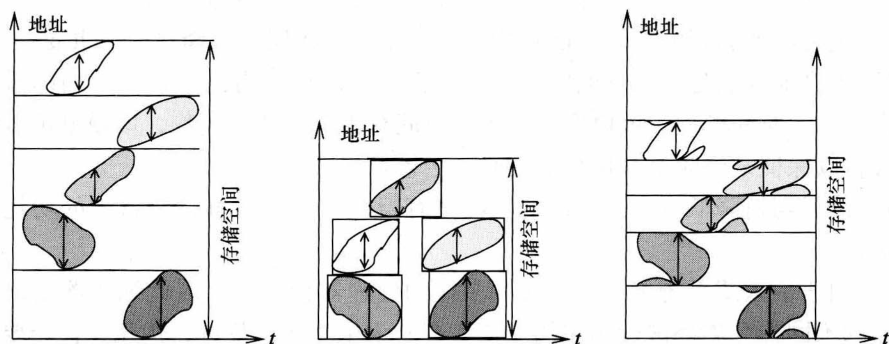



图7.12 未折叠数组（左图）、内部折叠数组（中图）以及改进的内部折叠数组（右图）

图7.12中图显示了**内部数组折叠**对内存使用的可能的优化方法，不在同一时刻被使用的数组可以共享同一段地址空间。图7.12右图显示了基于该折叠算法的演化[De Greef et al.，1997a]。该算法充分利用了数组内的有限集，可以使用更加复杂的地址算法以实现存储空间的节约。这两种折叠算法也可以合并使用。

Chung、Benini以及De Micheli［Chung et al.，2001］，［Tan et al.，2003］对其他上层转化算法进行了分析，并对编译器在相关领域的优化做出了卓越的贡献。

尤其特别要提出的是内联函数与常规调用函数的替换。使用内联函数会提高代码的运行速度，但是与此同时也会增加最终执行的代码的长度。在SoC中，代码尺寸的提高会带来一系列的后续影响。传统的内联代码需要编程者指定哪个函数被定义为内联函数，但是对于片上系统而言，其指令存储器的空间有限，因此找到一种能够根据目标系统存储器大小对源码自动进行内联函数转换的技术就非常重要。Teich［Teich et al.，1999］、Leupers等人［Leupers and Marwedel，1999］、Palkovic［Palkovic et al.，2002］以及Lokuciejewski［Lokuciejewski et al.，2009］等人在相关技术方面进行了大量的研究。这些基础既可以继承如编译器中转换，也可以在源代码编译之前在源代码中进行转换。

## 7.3 用于嵌入式系统的编译器

### 7.3.1 简介

很明显，大多数程序员对于在PC上运行的32bit程序的优化和编译已经有了非常深入的了解。对于嵌入式系统而言，由于标准编译器通常非常便宜甚至是免费的，嵌入式系统也经常使用标准编译器进行编译。

然而，有些原因导致了嵌入式系统必须使用定制的编译器：

1）嵌入式系统的处理器架构有其特殊性，编译器可以使用这些特性产生出非常有效率的代码。编译器也不得不支持3.3.3节所描述的压缩技术。

2）能够高效运行的代码比能够用高效率编译出代码的编译器更加重要。

3）编译器在满足以及证明实时性约束方面可能会有帮助。首先，编译器有可能包含明确的时间模型，该模型可以用于时间行为的优化。例如，该模型可以锁定缓存线，以保证某些频繁被执行的代码不会被换出缓存。

4）编译器对降低嵌入式系统的功耗也有帮助。编译器可以对系统的功耗进行某些优化。

5）对于嵌入式系统来说，有多种多样的处理器指令集，因此编译器要对不同处理器的不同指令集都进行相应的支持。有时为了对指令集进行优化，甚至要重定向编译器。对于此类编译器，可以将指令集设置为编译器生成系统的输入。这些系统可以对指令集进行实验性的修改，并且对修改后生成的机器码进行观察。例如Tensilica工具［Tensilica Inc.，2010］就对上述需求提供了支持。

［Marwedel and Goossens，1995］的文章中对重定向编译器进行了相关描述。在Leupers［Leupers，1997］，［Leupers，2000a］等人所著著作中可以找到与优化相关的内容。在本节中，将会展示有关嵌入式处理器的相关编译技术。

## 7.3.2 高效节能编译

许多嵌入式系统是使用电池作为能量供应源的移动系统。与移动计算能力的不断提升相比，电池技术的发展显得相对缓慢［ITRS Organization，2009］。因此，对于新的应用而言，电池的续航能力成为了移动系统发展的瓶颈。

节能可以在系统的不同阶段进行，如设备工艺、电路设计、操作系统以及应用程序的算法，甚至芯片的制造工艺对其都有影响。从源代码到机器码的转换对系统节能也有帮助。7.2节的上层优化技术对降低功耗也有助益。本节将着眼于通过对编译器的优化实现对功耗的降低（通常称为低功率优化）。功耗优化基于功耗模型，在第5章已经介绍了若干功耗模型。通过使用功耗模型，下列编译器可以用于降低系统功耗的优化工作中：

1）**节能调度**：在不改变程序目的的前提下，可以改变程序中指令的执行顺序。通过改变指令的执行顺序，可以对指令总线的访问进行优化。这种优化可以通过编译器执行，因此无需对编译器做任何修改。

2）**节能指令选择**：通常来说，对于同样的源代码可以生成不同的指令序列。对于标准编译器而言，指令的数量或者代码执行的周期数被用作所编译出来的代码是拥有优良指令序列与否的标准。该序列标准可以被系统功耗标准代替。Steinke等人发现，通过使用某些可以节能的指令，可以降低系统功耗的百分比［Steinke，2003］。

3）**对成本函数进行替换**也可以用于其他的标准编译器优化，例如寄存器流水线、循环不变式代码移动等。此类优化也可以降低若干系统功耗。

4）**利用分级存储器体系**：如3.4节所述，较小的内存拥有更快的访问速度以及更低的功耗，因此如果使用分级存储器体系，那么可以节约大量的系统功耗。在Steinke［Steinke et al.，2002b］，［Steinke et al.，2002a］对编译优化所作的分析中，分集存储体系架构所节省的系统功耗最多。因此除了使用较大的后台存储器外，可以使用小的便签式存储器（Scratch-Pad Memories，SMP），这样对系统功耗更有益处。对较小范围内存进行寻址所需的时间与功耗都低于对大内存的寻址。编译器负责将变量与指令分配至便签式存储器，但是该方法需要更频繁地对变量进行访问，并且代码序列也需要映射至该地址范围。

## 7.3.3 基于内存架构的编译

### 7.3.3.1 基于SPM技术的编译

使用SPM的优势已经被明确证明［Banakar et al.，2002］，因此使用SPM可以作为内存分层结构中最突出的示例。编译器通常可以将内存对象映射至存储器的某一地址范围，因此源码通常来说必须进行注释。例如，内存段可以在源码中使用类似注记：

```
# pragma arm section rwdata = "foo" ,rodata = "bar"
```

在 paragma 后声明的变量"foo"将被映射至读写段中，常量"bar"将被映射至只读段中。链接命令接下来会将这些段映射至特殊的地址范围，其中也包括 SPM。这是用于 ARM 处理器的编译器语句 [ARM Ltd.，2009b]。使用这种方式进行标记对于程序员而言有些繁琐，对编码者而言，更希望有一种能够对经常访问的对象进行自动映射的方法，因此对应的优化算法也相应衍生。HiPEAC 提出了一种解决方法 [Marwedel，2007]，可用的 SPM 优化可被分为如下两类：

1）非覆盖式（或“静态”）内存分配策略：对于此类策略而言，当应用程序运行时，与其对应的内存对象始终驻留在 SPM 中。

2）覆盖式（或“动态”）内存分配策略：对于此类策略，内存对象在程序运行时会不停地在 SPM 中换入换出。除了在 SPM 以及某些速度较慢的内存中的对象迁移外，该分配方式是一种“编译器分配页”，这种分配方式与磁盘中的数据对象无关。

#### 7.3.3.2　非覆盖式分配

对于非覆盖式分配，可以考虑对函数以及 SPM 中的全局变量进行分配。为了该目的，每个函数以及每个全局变量都可以被视作函数对象，令：

1）$S$ 为 SPM 的大小。

2）$sf_i$ 以及 $sv_i$ 分别为函数 $i$ 以及变量 $i$ 的大小。

3）$g$ 保存每次进行 SPM 访问所需的功耗（即对慢速主存进行每次访问所需的功耗与进行 SPM 所需的功耗之间的差值）。

4）$nf_i$ 以及 $nv_i$ 分别表示函数 $i$ 与变量 $i$ 的被访问次数。

5）$xf_i$ 以及 $xv_i$ 定义为

$$xf_i = \begin{cases} 1 & \text{如果均能 } i \text{ 映射到 SPM} \\ 0 & \text{其他} \end{cases} \tag{7.1}$$

$$xv_i = \begin{cases} 1 & \text{如果变量 } i \text{ 映射到 SPM} \\ 0 & \text{其他} \end{cases} \tag{7.2}$$

目标为最大的系统功耗值：

$$G = g\left( \sum_i nf_i \cdot xf_i + \sum_i nv_i \cdot xv_i \right) \tag{7.3}$$

以及相关的空间约束

$$\sum_i sf_i \cdot xf_i + \sum_i sv_i \cdot xv_i \leqslant S \tag{7.4}$$

该问题被称作渐缩问题。标准渐缩算法可以用于选择分配至 SPM 中的对象，然而式（7.3）与式（7.4）依旧有整形线性规划（Integer Linear Programming，ILP）问题（见附录 A），并且 ILP-solvers 也可以使用。$g$ 是目标函数的常数因子，并且并不需要解决 ILP 问题。相应的优化可以通过使用前通优化实现（见图 7.13）。

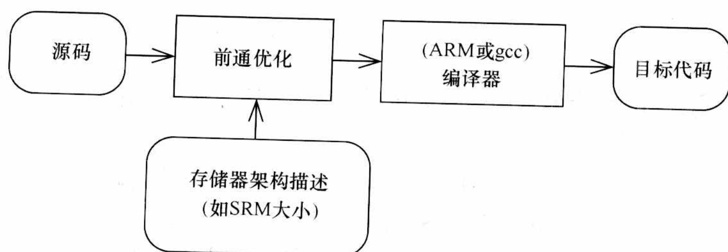



图 7.13 前通优化

优化对函数的地址以及全局变量有相应的影响。编译器通常允许在源码中对这些地址进行手动分配，因此编译器本身没有任何必需的变化。前通优化的优点是可以与编译器在多种不同的目标处理器上使用，因此没有必要对大量的制定目标编译器进行修改。

该模型可扩展至不同的方向：

1）**基本块的分配**：刚刚描述的方法只允许将整个函数与变量分配至 SPM 中。其结果是，如果函数或变量所需的空间较大，则 SPM 将会有小块的空间无法使用，因此尝试对分配至 SPM 的对象进行粒度缩减。一种很自然的选择是考虑将基本块作为内存对象。此外，还考虑一组相邻的基本块，相邻的含义为一组由处理器所定义的相邻指令地址空间，将这种相邻的块称为多重块。图 7.14 显示了 3 个由基本块 BB1、BB2 以及 BB3 组成的多重块 M12、M23 以及 M123。

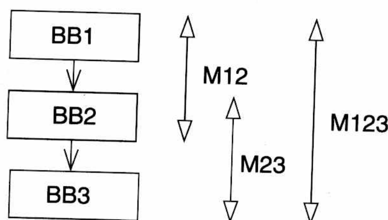



图 7.14 基本块以及多重块

ILP 模型可以被扩展如下：

① 令 $sb_i$ 以及 $sm_i$ 分别为基本块 $i$ 以及多重块 $i$ 的尺寸；

② 令 $nb_i$ 以及 $nm_i$ 分别为基本块 $i$ 以及多重块 $i$ 的访问次数；

③ 令 $xb_i$ 以及 $xm_i$ 定义为

$$xb_i = \begin{cases} 1 & \text{如果基本块 } i \text{ 映射到 SPM} \\ 0 & \text{其他} \end{cases} \tag{7.5}$$

$$xm_i = \begin{cases} 1 & \text{如果多重块 } i \text{ 映射到 SPM} \\ 0 & \text{其他} \end{cases} \tag{7.6}$$

所能获取的最大增益为

$$G = g\left(\sum_i nf_i \cdot xf_i + \sum_i nb_i \cdot xb_i + \sum_i nm_i \cdot xm_i + \sum_i nv_i \cdot xv_i\right) \tag{7.7}$$

相应的约束为

$$\sum_i sf_i \cdot xf_i + \sum_i sb_i \cdot xb_i + \sum_i sm_i \cdot xm_i + \sum_i sv_i \cdot xv_i \leqslant S \tag{7.8}$$

$$\forall \text{基本块}\ i: xb_i + xf_{fct(i)} + \sum_{i' \in \text{multiblock}(i)} xm_{i'} \leqslant 1 \tag{7.9}$$

第二种约束保证每个基本块到 SPM 的映射只有一次，而非封闭函数以及多重块的映射。

Steinke [Steinke et al., 2002b] 等人进行了基于该模型的相关实验。对于某些基准测试程序而言，尽管能够放入 SPM 的数据仅仅占据了总代码量的一小部分，但是最多还是可以降低 80% 的功耗。基于冒泡排序所得的测试结论如图 7.15 所示。

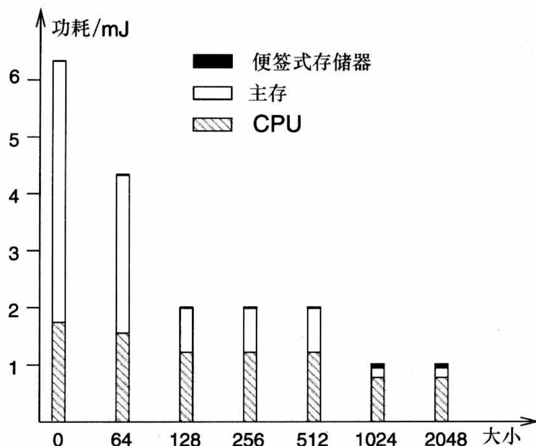



图 7.15　基于编译器 SPM 映射的冒泡排序所降低的功耗

很明显，在假定系统供给电压恒定的前提下，SPM 有助于降低系统主存所带来的功耗。由于降低了处理器等待周期，所以对于处理器功耗的降低也有帮助。

2) **分块存储器** [Wehmeyer and Marwedel, 2006]：较小的内存拥有更快的运行速度以及更低的系统功耗，因此将处理器分为若干个小处理器是有意义的。ILP 模型很容易就可以扩展用于若干存储器的模型，在此情况下，不区分内存对象的不同类型（函数、基本块、变量等）。索引 $i$ 用于只带任意内存对象，令：

① $S_j$ 作为存储器 $j$ 的大小；

② $s_i$ 为对象 $i$ 的大小；

③ $e_j$ 为对存储器 $j$ 每次访问的功耗；

④ $n_i$ 为访问对象 $j$ 的次数；

⑤ $x_{i,j}$ 定义为

$$x_{i,j} = \begin{cases} 1 & \text{如果对象}\ i\ \text{映射到存储器}\ j \\ 0 & \text{其他} \end{cases} \tag{7.10}$$

为了最大化降低系统功耗，需要降低系统的整体功耗，因此目标为尽量减少

$$C = \sum_j e_j \sum_i x_{i,j} \cdot n_i \tag{7.11}$$

考虑如下约束：

$$\forall j: \sum_i s_i \cdot x_{i,j} \leq S_j \tag{7.12}$$

$$\forall i: \sum_j x_{i,j} = 1 \tag{7.13}$$

特别是对于不同的内存需求而言，进行内存分块对于系统效率及功耗而言是有其积极意义的。被频繁访问的存储区域被称作应用程序的工作集，拥有较小工作集的程序可以分配较小的高速存储器，反之拥有较大工作集的程序需要分配更大的存储空间。因此，对内存进行分区的一个关键优势就是该存储器可以适应当前工作集的不同大小。

此外，没有使用的存储器可以关闭用来节省功耗，然而考虑到只有对内存的访问才会产生“动态”功耗。此外，即使存储器空闲，也会有功耗产生，但暂时不去考虑这种功耗。因此，关闭存储器所节省的功耗并没有在式（7.11）与式（7.12）中有所体现。

3）**链接/加载时分配内存**［Nguyen et al.，2005］：在编译时对某一SPM进行代码优化有其不利的一面：当代码运行于某些不同架构的处理器上时，如果这些处理器的SPM容量不同，其运行效率有可能会降低。要尽量避免不同的执行文件在不同的处理器上运行，因此对独立于SPM大小而运行的可执行文件更感兴趣。如果在链接时进行优化，则上述场景则是可行的，计算的方法是将访问次数除以变量在编译时的大小所得的比值以及存储该值与其他可执行文件中的相关变量的信息。在装载时，OS需要获取SPM的大小，然后对代码进行对应的优化，使其尽可能多在SPM中进行变量的分配。

4）**在栈中分配**：为了真正的降低功耗，所有经常访问的内存对象都必须分配至某些较小的内存中。栈也是需要考虑的一种，否则对栈的访问将会限制系统的整体效率。目前至少有两种方式用于对栈的访问：Steinke［Steinke et al.，2002b］通过使用栈容量分析工具，可以计算出最差情况下栈的大小。如果栈足够小，则可以在SPM中进行分配。Avissar等人［Avissar et al.，2002］提出了一种对栈进行分割，使其可以对频繁访问的元素和不频繁访问的元素进行区分的方法。访问次数不频繁的元素被放置在慢速的主存中，而访问次数频繁的元素放置在SPM中。调度器需要知道不同内存中栈的指针。为了降低对每个函数调用进行两个栈指针更新的总开销，堆栈在分割时要注意不要对“小”函数进行相应分配。

5）**在堆中进行分配**［Dominguez et al.，2005］：之前关于对栈的分配的讨论也适用于堆，频繁被访问的堆元素也应被分配至运行更快的存储器中。可以使用堆尺寸分析器去计算堆的上界，较小的被频繁访问的堆可以完全分配至SPM中，然而堆的尺寸往往大于SPM。Dominguez等人提出了第二种方法，这种方法认为，程序可以分割成若干个区域，各个区域可以由程序指针进行限定，这些程序指针对于这种方法来说极为关键。程序指针可以定义为［Udayakumaran et al.，2006］：

“（i）每个进程的开始即结束；（ii）只位于每个循环开始前及每个循环结束后（甚至是内部的嵌套循环）；（iii）每个 if 语句的 then 部分和 else 部分的开始及结束以及 if 语句的开始和技术；（iv）程序的 switch 语句的每个 case 分支的开始及结束以及整个 switch 语句的开始及结束”。

在［Dominguez et al.，2005］中提到，某些空间在 SPM 中始终保持是可用的，并且每个时刻只有一个代码区可以进入，如果有需要，堆元素在移入或移出 SPM 时会进行相应的复制。如果该复制完成，则指向堆元素的指针始终保持有效。

6）**考虑到对时间可预测性的影响**［Wehmeyer 以及 Marwedel，2006］：无论访存操作发生于高速内存还是低速内存，大多数的 SPM 分配算法都在编译时进行。因此，相较于对缓存的访问而言，对存储器的访问速度进行预测是可行的。因此，基于 SPM 系统的最坏情况下访问时间通常优于基于缓存的系统。

#### 7.3.3.3 上覆分配

大型的应用有可能需要多个热点（多重区域代码包括密集计算型循环），非叠加的方法无法为该上下文提供最好的系统效能。对于这些应用而言，SPM 可以用于每个热点，这就需要代码能够在内存架构层面进行自动迁移。有若干种方法可以用于上覆分配：

1）**平铺大型数组**［Kandemir et al.，2001］，［Chen et al.，2006］：大型数组由于尺寸无法满足 SPM 要求，所以使用 SPM 时可能会产生问题。目前所提到的算法不能将数组的某个子集放置到 SPM 中。Kandemir 提出了一种算法可以用于该种场景下的 SPM 分配，该技术可以将数组分片复制至 SPM 中。在［Kandemir et al.，2001］中提到，平铺可以无条件使用。在［Chen et al.，2006］中，作者提出了不要对不规则的数组进行平铺访问，因为对这种数组进行平铺访问会导致效率低下。

2）**多重层次分配**［Brockmeyer et al.，2003］：大容量存储器与小容量存储器之间的访问速度差距正在不断拉大，因此提出多重层次分配是有意义的。IMEC 的 MHLA（架构级别存储器分配）工具尝试找出一种能够在不同存储器级别进行变量分配的方法。MHLA 在程序进入循环前自动选择可以被复制至快速存储器中的数组子集。该工具的最新版本由 Mnemee project［Mnemee project，2010］进行了相应的设计。

3）**基于区域的内存对象迁移**［Udayakumaran et al.，2006］：该方法基于对存储器的分区以及程序指针。每个程序指针需要考虑将哪个变量移入或移出 SPM（代码可以作为一种变量进行模拟）。

4）Verma 的方法［Verma and Marwedel，2004］与 Udayakumaran 的方法类似，然而选择要复制的内存对象基于全局 ILP 模型，而非本地数据。

#### 7.3.3.4 多线程/进程

上述方法始终限定于用于单处理器或单线程下的场景。对于多线程而言，将对象移入或移出 SPM 需要在进程的上下文中进行考虑。Verma［Verma et al.，2005］

提出了三种不同的方法：

1）对于第一种方法，在给定时间内，只有一个处理器拥有 SPM 空间。在每次上下文切换时，被抢占的进程所占据的 SPM 空间的相关信息被保存，并且该信息在进程再次被执行时恢复，该方法被称为**保存/恢复法**。但是在 SPM 容量较大的情况下，由于复制过程需要消耗大量的时间以及能量，该方法显得不是特别有效。

2）对于第二种方法，可以将 SPM 划分为若干个区域，以供不同的进程使用。划分区域的粒度由优化级别来决定，SPM 在初始化时就被填满。这里不需要编译器进行进一步的编译器控制复制，因此该方法被称作**无需保存法**。这种方法只有在 SPM 大到足以包含若干进程所需的存储空间的情况下才有意义。

3）第三种方法被称作混合法：这种方法将 SPM 分割为若干的小块，这些小块可以供不同的进程使用，此外再留有一个较大的存储区间，以供所有进程使用。这两种区间的空间由优化级别来决定。

Verma 的方法需要在编译时就获知一组进程的固定信息。接下来是进程在系统中的运行以及消亡。Pyka 等人［Pyka et al.，2007］描述了集成在操作系统中的 SPM 存储管理器（SPM Memory Manager，SPMM），该组件用于在运行时对 SPM 进行分配。与之前的算法相比，Pyka 的方法允许在对库进行预编译时将库中的代码放入 SPM 中。不幸的是，Pyka 的算法需要一个额外的中间层。尽管这个额外的中间层增加了系统的总开销，但是所获得的 4 路相连缓存可以降低系统 25% ~35% 的总功耗。

如果系统的存储器管理单元（Memory Management Unit，MMU）是可用的，则上述的额外开销是可以避免的。Egger 等人［Egger et al.，2006］通过利用 MMU 开发出了相应的技术：在编译时，无论代码段是否可以通过在 SPM 中进行分配而获得受益，代码段都会进行相应的分类。受益的代码被存储在某一虚拟地址空间中，最开始该区域不会被映射到物理内存上，因此当代码第一次被访问时会发生缺页异常。缺页异常会激活 SPMM，SPMM 会对 SPM 空间分配，并对虚实地址转换表进行更新。

通常来说，对 SPM 的操作需要工具的支持，但更需要良好的系统设计。如果没有这种工具的支持，则可以使用缓存。未来的系统有可能同时包含缓存以及 SPM。

### 7.3.4　调和编译器以及时序分析

如今所有可用的编译器几乎都不包含时序模型，因此实时软件的开发者通常不得不遵循软件迭代法：通过该种编译器所得的软件是无法得知相关的时间信息的。若想获知相关的时间信息，则需要某些工具（如 aiT［Absint，2010］）对所生成的代码进行分析。如果时序约束不满足原定要求，则需要对编译器运行所需的输入进行改变，并再次进行编译，重复该过程直到满足原定时序约束要求，称其为基于反

**复试验的实时软件开发**。这种方法存在若干问题：首先，所需的迭代数在设计之初是未知的；其次，这种方法中使用的编译器是“优化”的，但是除了目标代码的大小外，对程序其他部分进行精确的评估是不可能的。因此，编写编译器的程序员只能期望这些“优化”对相关代码的质量有着积极的影响。由于现代处理器的行为非常复杂，几乎没有证据能够直接证明这种期望。最后，这种基于**反复试验的实时软件开发**需要设计者找出能够满足实时性约束的恰当的编译器输入。

如果编译器中包含了时序分析模块，就可以避免使用这种基于反复试验的方法。多特蒙德理工大学的 WCC 就包含了该模块，旨在用于最差情况下执行时间开发的编译器。独立于现有产品，开发一个完整的时序分析模块无疑是极大的资源浪费。因此 WCC 基于 aiT 的时序分析模块与 TriCore 公司试验用的编译器进行了整合。图 7.16 显示了 WCC 的整体架构。

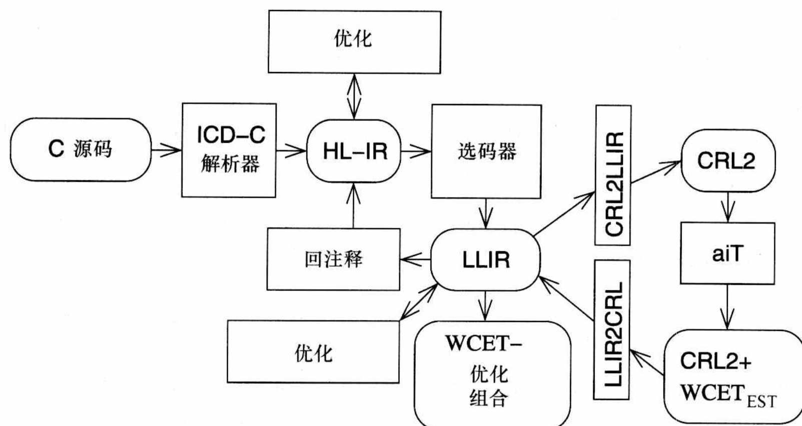



图 7.16　能够获知最坏情况下执行时间的编译器 WCC

WCC 使用 ICD-C 编译器架构［ICD Staff, 2010］读取 C 源代码并进行语义分析，该源代码会被转化为“与平台无关的中间表示层”（HL-IR）。HL-IR 对源码进行了抽象表示，各种优化方法都可以用于 HL-IR。优化后的 HL-IR 被传送至选码器。选码器将这些源码的操作映射至对应的机器指令。WCC 至今为止一直专注于对 Infineon TriCore 架构进行支持。TriCore 的指令集由低级中间表示层 LLIR 进行表示。为了估算 $WCET_{EST}$，LLIR 会通过 aiT（用于进行 LLIR 到 CRL2 的转换）转换至 CRL2。aiT 会根据给定的机器码生成 $WCET_{EST}$。该信息会转化回 LLIR 模式（使用 CRL 至 LLIR 转换）。WCC 在优化时会使用该信息进行 $WCET_{EST}$ 的估算。对 LLIR 级别进行优化而言，该方法简单、直接，然而许多的优化过程需要在 HL-IR 级别进行。在该级别进行 $WCET_{EST}$ 优化需要从 LLIR 到 HI-IR 级逆向注解，ICD-C 包括了这种逆向注解。

WCC 用于研究在编译器中降低 $WCET_{EST}$ 对于优化的影响。这些结论中包括采

用这一目标寄存器分配的影响的研究［Falk，2009］。所得出的结论有些戏剧化，如图 7.17 所示。

图 7.17　通过使用 WCET-aware 寄存器分配后所降低的 $WCET_{EST}$ 图

通过使用 WCC 的已知 WCET 寄存器分配，可以将原始的 $WCET_{EST}$ 的平均消耗降低 68.8%。但与原始的 $WCET_{EST}$ 相比，效率最多会降低 24.1%。Lokuciejewski 等人［Lokuciejewski and Marwedel，2010］对若干优化组合的效果进行了分析。通过使用相同的性能分析工具，Lokuciejewski 发现相比于原始的 $WCET_{EST}$，效率最多有 57.1% 的提升。

### 7.3.5　DSP 编译

DSP 的特性已经在 3.3.3.1 节有了相应的描述，编译器应充分利用在第 5 章提到过的准则进行代码优化，可以使用地址生成单元作为示例以证明这些技术。生成“空”地址的可能性对于变量如何在内存中进行布局有着重要的影响。图 7.18 给出了相应的示例。

| 地址 | 变量 | 指令 | 访问 |
|---|---|---|---|
| 0 | $a$ | LOAD $A$, 1 | ; $b$ |
|  |  | $A$+=2 | ; $d$ |
| 1 | $b$ | $A$-=3 | ; $a$ |
|  |  | $A$+=2 | ; $c$ |
| 2 | $c$ | $A$++ | ; $d$ |
| 3 | $d$ | $A$-- | ; $c$ |

| 地址 | 变量 | 指令 | 访问 |
|---|---|---|---|
| 0 | $b$ | LOAD $A$, 0 | ; $b$ |
|  |  | $A$++ | ; $d$ |
| 1 | $d$ | $A$+=2 | ; $a$ |
|  |  | $A$-- | ; $c$ |
| 2 | $c$ | $A$-- | ; $d$ |
| 3 | $a$ | $A$++ | ; $c$ |

图 7.18　内存布局对比

假设对于某些基本块，变量 $a \sim d$ 的访问基于某个固定顺序（$b$，$d$，$a$，$c$，$d$，$c$）。使用寄存器间接寻址来进行这些变量的访问。首先，将地址 $b$ 放入地址寄存器（见图 7.18 左图）。由于目前所关注的是地址生成，与变量 $b$ 相关的机器指令没有在图 7.18 中标明，因此用于访问下一个变量（$d$）所生成的地址将在稍后进行说明。假设寄存器 A 只能放入一个地址，$A$ 必须更新为指向变量 $d$ 的地址，这需要将当前寄存器的值加 2。接下来，忽略读取变量的指令，并且立即考虑对 $a$ 的访问。对于该操作，必须将其进行减 3 操作，并且在下次访问时再加 2。假设自增以及自减的范围被限定为 ±1，只有图 7.18 中的最后两个访问可以通过这些操作进行。总

而言之，需要在指令中进行地址的计算。

与之相反的是，对于图 7.18 右图的布局，4 个地址的计算是可以并行在主数据通路上执行的自增与自减操作。对于偏移大于 1 的操作而言，只需要进行两个周期的地址计算。在这里也没有写出使用变量的指令。

如何生成这种智能的内存布局？算法通常从访问图（见图 7.19）中开始进行。

这些访问图的每一个变量都与一个节点相对应，并且每一对相邻的变量都有一个连接，这些连接的权值与相邻变量的访问次数有关。

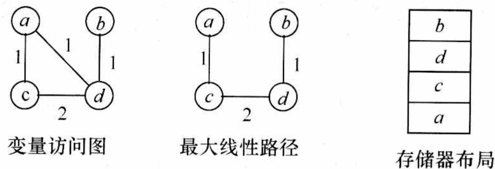



图 7.19　对单地址寄存器 $A$ 的内存分配访问顺序（$b$，$d$，$a$，$c$，$d$，$c$）

由高权值的连接所系的变量应该被分配至相邻的内存区域。通过这种方式保存的地址计算次数与其权值相等。例如，如果 $c$ 与 $d$ 分配至相邻的区域，那么接下来的两次访问就可以仅仅通过自增和自减操作进行。

进行内存分配的总的目标就是找出一种内存中变量的线性次序，以期能够在内存寻址中尽量使用自增以及自减操作。这与找出访问图中的最大权值变量的先行路径对应。不幸的是，在图中最大权值路径问题是一个非完全多项式问题，因此通常使用试探法来生成该路径 [Liao et al.，1995b]，[Sudarsanam et al.，1997]。通常这些方法都基于 Kruskal 生成树。下面为 Liao 算法：

1）对访问图 $G=(V, E)$ 的连接进行基于权值的排序。

2）构造一个新图 $G'=(V', E')$，并使 $G'=G$、$E'=0$。

3）选择图 $G$ 最高权值的连接 $e$，如果该连接不会导致 $G'$ 以及任意 $G'$ 的节点的周期大于 2，则将其加入节点 $E'$ 中，否则丢掉 $e$。

4）回到第 3 步，选择 $G$ 中权值尽可能长的连接，并选择（$|V|-1$）至 $G'$ 中。

隐式的，所有节点都假设由权重为 0 的连接所系，这确保了即使图的某个部分没有连接，该算法也能继续运行。变量在存储器中的顺序与沿线性路径所生成的变量顺序相对应。

基于图 7.19 的该算法的应用示例如图 7.20 所示。

连接（$c$，$d$），基于其权重，该连接为第一个加入图 $G'$ 中的连接。所有权重为 1 的连接中，其顺序是可以任意排布的。假设（$a$，$c$）接下来加入图中（见图 7.20 中图），（$a$，$d$）有可能是接下来需要考虑的连接。该链接加入 $G'$ 中将会导致产生一个循环，所以该链接被丢弃，最终（$b$，$d$）加入图 $G'$。最终，图中加入了有 3 个连接的 4 个节点后，算法停止运行。

该算法仅仅能够对简单的场景进行处理。Leupers 以及 Marwedel [Leupers and Marwedel, 1996] 在等权启发方法上发表了若干的论文，该方法基于更复杂的场景对该算法进行了扩展，例如：

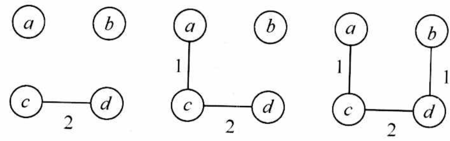


图 7.20 Liao 算法的步骤序列

1) $n>1$ 地址寄存器 [Leupers and Marwedel, 1996]；

2) 依旧使用 AGU 中提到的修改寄存器 [Leupers and Marwedel, 1996], [Leupers and David, 1998]；

3) 对数组进行功能扩展 [Basu et al., 1999]；

4) 更大的自增以及自减的范围 [Sudarsanam et al., 1997]。

该方法对上面所提到的内存分配进行了代码尺寸以及实时代码生成的扩展，还有一些优化算法利用了 DSP 的架构特性，例如：

1) 多重存储器组 [Sudarsanam and Malik, 1995]；

2) 异构寄存器文件 [Araujo and Malik, 1995]；

3) 模寻址 [Quilleré and Rajopadhye, 2000]；

4) 并行指令集 [Leupers and Marwedel, 1995]；

5) 多种操作模式 [Liao et al., 1995a]。

Leupers 描述了其他的优化技术 [Leupers, 2000a]。

### 7.3.6 多媒体处理器的编译

为了完全的支持 3.3.3.1 所描述的打包数据类型，编译器必须能够在循环中对这些包自动转换，这样就可以生成更有效率的软件。在编译器中实现该功能是一件非常具有挑战性的任务。编译算法用于打包数据类型的操作是基于超级计算机的向量算法所作的扩展，一些用于多媒体以及 SIMD 的短向量扩展算法在 [Fisher and Dietz, 1998], [Fisher and Dietz, 1999], [Leupers, 2000b], [Krall, 2000], [Larsen and Amarasinghe, 2000] 相关著作中进行了描述。

用于 M3-DSP 的自动并行循环需要向量化技术，该技术可以获得显著的速度提升（与顺序操作相比，见图 7.21）[Lorenz et al., 2002], [Lorenz et al., 2004]。对于应用程序 dot_product_2 而言，由于向量的尺寸过小，以致其可以达到非常快的速度。此外该程序不需要进行数据向量化。如果向量操作与零开销循环指令相结合，则最多可以降低 94% 的系统时钟周期开销。

由于增加了 SIMD 扩展的处理器数量，用于 SIMD 指令的编译备受关注 [Ren et al., 2006], [Nuzman et al., 2006]。特别是用于 Cell 处理器的编译再次唤起了人们对于这种编译技术的兴趣（例如参见 Eichenberger 等人的著作. [Eichenberger et al., 2005]）。此外，编译器的编写者所研究的短向量指令技术可以用于 Pentium® 兼容处理器中 [Gerber et al., 2005]。对于这个充满活力的研究领域，

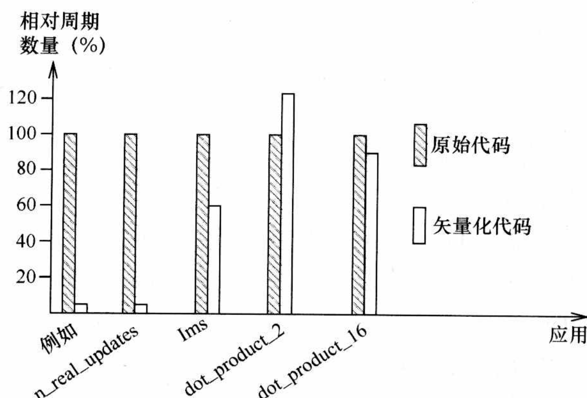



图7.21 通过对M3-DSP向量化后降低的周期数

想提供一个全景式的预览也是很难做到的。

### 7.3.7 用于VLIW处理器的编译器

VLIW架构需要对编译器进行特殊的优化：

1）TMS 320C6xx编译器优化的关键是能够在编译时对某一功能单元执行某种操作。由于该处理器有两条数据通路（见图3.28），这意味着具体的操作需要分成两个对应的子集［Jacome and de Veciana，1999］，［Jacome et al.，2000］，［Leupers，2000c］，并且包含对应的寄存器配置文件。

2）VLIW处理器通常有若干分支延迟槽。对于VLIW处理器而言，由于每个分支指令槽不能包含完整的指令包，在进行分支处理时VLIW处理器经常会获得比其他处理器更多的时间开销。例如，对于TMS 320C6xx而言，其分支延迟的惩罚时间一般为5×8=40个指令。为了避免如此大的惩罚，大部分VLIW处理器支持对大量的状态寄存器进行预测执行。预测执行可以用于对效率要求较高的短If语句中，对于长if语句而言，由于其可以在then以及else分支中互斥运行，状态预测可以获得更高的效率。可以找到适当的优化技术在两种if语句的实现方法中进行权衡［Mahlke et al.，1992］，［August et al.，1997］，［Leupers，1999］。

3）由于分支延迟的惩罚，inline函数是另一种在VLIW处理器中常用的优化方法。

4）在对Intel公司的IA-64 EPIC架构处理器进行编译器设计的工作方面，研究人员投入了大量的精力（见［Dulong et al.，2001］）。由于该架构的独特性，需要特殊的优化技术。

5）Trimaran用于在指令级别并行性以及VLIW和EPIC架构方面的编译器技术研究平台［Trimaran，2010］。

### 7.3.8 用于网络处理器的编译器

网络处理器是一种新型的处理器，该处理器对高速 Internet 应用程序进行了优化。这种处理器包含了对数据流信息中的位域进行访问及处理的指令集。通常而言，由于这种应用程序对于数据吞吐量要求极高，通常使用汇编语言进行编程。然而，越来越复杂的网络协议要求进行此类编译器的设计时需要尽量考虑对具体网络组件的支持。Falk、Wagner 等人［Falk et al.，2006］对具体的 bit 级细节进行了分析。

### 7.3.9 编译器的产生、重定向以及设计空间的研究

在设计人类有史以来第一个编译器时，编译器的设计工作是一个完完全全的手工操作过程。在此期间，产生编译器所需的某些步骤慢慢地可以通过某种工具自动进行，例如 lex 和 yacc 以及更多的类似工具（见［Johnson，2010］）提供了一些标准手段用于解析源码。生成机器指令是编译过程的其中一个步骤。例如，类似olive［Tjiang，1993］的树模式匹配即可以用于此类工作。尽管有着各种工具可以使用，编译器设计还是无法彻底自动进行，然而有很多尝试进行重定向编译器设计的手段，可以通过不同种类的可重定向能力进行分类：

1）开发者重定向能力：在此情况下，编译器的专家负责对新的指令集进行编译器重定向。

2）用户重定向能力：在此情况下，使用者负责对编译器进行重定向。这种方法通常而言更具挑战性。

更多的有关编译器的重定向以及在设计空间进行研究的信息可以在 Leupers 以及 Marwedel［Leupers and Marwedel，2001］的书籍中获得。包含上述功能的商业产品可以参见 Tensilica 公司相关产品［Tensilica Inc.，2010］。

## 7.4 电源管理以及温度管理

### 7.4.1 动态电压调节

某些嵌入式处理器支持动态电源管理以及动态电压调节（Dynamic Voltage Scaling，DVS），可以使用这些特征用于进行系统优化。通常来说，优化是随着编译器生成代码的过程进行，但使用上述两种方式进行的优化则需要对系统的所有任务有一个全局性的总览，包括这些任务的依赖性以及时间片信息等。

接下来的示例［Ishihara and Yasuura，1998］证明了动态电压调节在系统优化方面的潜力。假设有一个能够在 3 种不同电压（2.5V、4.0V 以及 5.0V）下运行的处理器，假设处理器运行于5.0V 电压下，每个时钟周期的系统功耗是 40nJ，式

（3.14）可以用于计算其他电压下的系统功耗（见图7.22，2.5nJ是一个约值）。

此外，假设任务需要在25s内运行$10^9$个时钟周期。在图7.23～图7.25中可见，该任务有若干种执行的形式。在使用最大电压的情况下（示例a，见图7.23），处理器有5s的时间处于空闲状态（假设在此期间处理器功耗为0）。

| $V_{dd}$/V | 5.0 | 4.0 | 2.5 |
|---|---|---|---|
| 每个周期的能耗/nJ | 40 | 25 | 10 |
| $f_{max}$/MHz | 50 | 40 | 25 |
| 执行周期/ns | 20 | 25 | 40 |

图7.22 拥有DVS的处理器特征

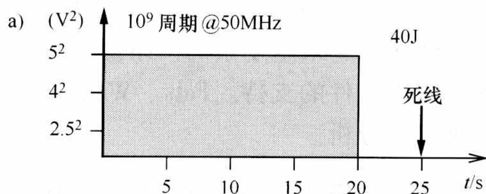



图7.23 第一种电压计划表

另一种场景下（示例b），在最开始使用全速运行处理器并在任务运行完毕后，将系统电压降到最低（见图7.24）。

最终，让处理器运行在能够满足任务死线约束的最低电压中（示例c，见图7.25）。

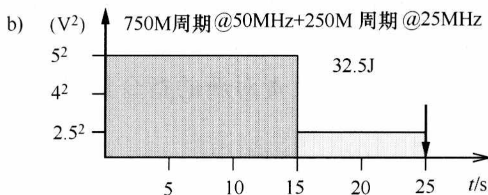



图7.24 第二种电压计划表

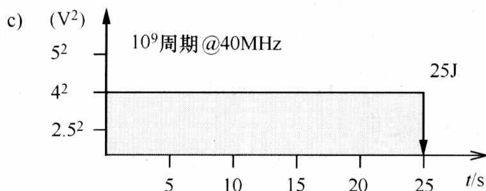



图7.25 第三种电压计划表

相应的功耗可以通过下式进行计算：

$$E_a = 10^9 \times 40 \cdot 10^{-9} = 40(\mathrm{J}) \tag{7.14}$$

$$E_b = 750 \cdot 10^6 \times 40 \cdot 10^{-9} + 250 \cdot 10^6 \times 10 \cdot 10^{-9} = 32.5(\mathrm{J}) \tag{7.15}$$

$$E_c = 10^9 \times 25 \cdot 10^{-9} = 25(\mathrm{J}) \tag{7.16}$$

可以发现，在第三种场景即系统电压稳定在4V的情况下，系统功耗最低。接下来将使用术语可变电压处理器用于描述允许使用任意电压的处理器。能够支持任意电压的可变电压处理器价格非常昂贵，因此实际的处理器仅仅支持若干个固定的电压。

在上述事例中进行观察所得出的结论可以用如下声明进行描述，Ishihara以及Yasuura在其论文中对这些声明进行了相应的证明：

1）如果处理器在某个可用电压下提前完成了任务，那么系统功耗可以降低[⊖]。

---

⊖ Ishihara与Yasuura的论文中明确地使用了隐含假定的论点。

2）如果某个处理器使用单一电压 $V_s$ 并且在任务 $T$ 死线到来的时刻将任务 $T$ 运行完毕，那么在电压 $V_s$ 下运行任务 $T$ 可以得到最低的系统功耗。

如果处理器可以使用若干离散的电压，那么电压计划表可以选择使用两个相邻的理想电压值 $V_{ideal}$。这两个电压值可以使系统功耗为最低功耗[⊖]。

它可以用于对某些任务指定其运行电压。接下来考虑将某个电压分配给某组任务，将使用如下符号：

1）$N$：任务的数量。

2）$EC_j$：任务 $j$ 的执行周期数。

3）$L$：目标处理器可运行的电压数。

4）$V_i$：第 $i$ 个电压，$1 \leqslant i \leqslant L$。

5）$F_i$：基于电压 $V_i$ 的时钟频率。

6）$D$：所有任务必须运行完毕的全局死线。

7）$SC_j$：在执行任务 $j$ 时的平均开关电容量（$SC_i$ 包括实际电容量 $C_L$ 以及开关次数 $\alpha$ [见式（3.14）]

电压调节问题可以用整数线性规划（Integer Linear Programming，ILP）问题描述。为达到该目的，引入变量 $X_{i,j}$，该变量为在指定电压下执行任务的周期数。

基于 ILP 模型的假设包括：

1）一个可以在有限个离散数量电压下运行的目标处理器。

2）进行电压转换所需的时间可以忽略不计。

3）每个任务运行所需的最多周期数是已知的。

使用这些假设，ILP 问题可以论证如下：

最小化

$$E = \sum_{j=1}^{N} \sum_{i=1}^{L} SC_j \cdot X_{i,j} \cdot V_i^2 \tag{7.17}$$

依赖于

$$\forall j: \sum_{i=1}^{L} X_{i,j} = EC_j \tag{7.18}$$

以及

$$\sum_{j=1}^{N} \sum_{i=1}^{L} \frac{X_{i,j}}{F_i} \leqslant D \tag{7.19}$$

目标是找到每个任务 $j$ 在某一电压 $V_i$ 执行时所需的周期数 $X_{i,j}$。通过上述声明，没有任何任务需要两个以上的电压值。使用该模型，Ishihara 以及 Yasuura 指出如果任务使用的电压越高，则系统效率通常越高。如果有大量的系统空闲时间，可以通过尝试不同的电压值找到对系统效率最优帮助的电压值。然而对处理器而言，能

---

⊖ 在最初的论文中并没有将这些考虑在内。

够提供4种不同的电压值就足以处理不同的场景。

在许多示例中，程序的运行速度都快于预测的最坏情况下执行时间。但上述的算法并不能对这种情况进行描述。在使用检查点对实际的最坏情况下执行时间与预测执行时间相比对并通过该信息降低系统电压时［Azevedo et al.，2002］，可以将此限制移除。另外，多速率任务图下的电压调节也被提出［Schmitz et al.，2002］。DVS可以作为一个优化手段与其他优化方法一起使用，例如体偏压［Martin et al.，2002］。体偏压是一种用于降低漏电流的技术。

### 7.4.2 动态电源管理

为了降低系统功耗，也可以使用3.3.3节所述的电源节能状态。使用动态电源管理（Dynamic Power Management，DPM）的根本问题是在什么时候进入电源节能状态？一个直接而简单的方法是使用一个定时器转换至节能态。使用更加复杂的随机过程模型计算系统闲置时间可以更准确地预测系统的使用状况。基于指数分布的模型已经被证明是不正确的。更新的理论描述了更加精确的系统模型［Simunic et al.，2000］。

有关电源管理的全面讨论的论文已被发表（参见［Benini and De Micheli，1998］，［Lu et al.，2000］）。这些更先进的算法将DVS以及DPM一起使用用于系统节能的优化［Simunic et al.，2001］。

分配DPM的电压以及对DPM的转换时间进行计算可能在嵌入式软件优化过程中需要分两个步骤进行。

功耗管理与散热管理紧密相连。散热管理依赖于运行时的系统温度信息。这些信息用于处理系统的发热，并使用系统制冷机制对系统进行冷却。在系统散热管理中，控制风扇转速是最简单而有效的手段。此外，如果系统温度超过了阈值，则系统有可能彻底关闭。更先进的系统有可能会降低系统的时钟频率以及电压。对于多核系统而言，任务有可能在不同的核上运行。在所有这些情况下，要对系统运行时的“温度”进行客观评价，评价所得的结论会对系统运行产生深刻的影响。Merkel等人［Merkel and Bellosa，2005］以及Donald等人［Donald and Martonosi，2006］发表了如何避免系统过热方面的著作。

## 7.5 思考题

1. 考虑下面这样一个程序：

```
1  #include <stdio.h>
2  #define DATALEN 15
3  #define FILTERTAPS 5
4  double x[DATALEN] = { 128.0,130.0,180.0,140.0,120.0,
```

```
5                      110.0,107.0,103.5,102.0,90.0,
6                      84.0,70.0,30.0,77.3,95.7 };
7  const double h[FILTERTAPS] = { 0.125, -0.25,0.5, -0.25,0.125 };
8  double y[DATALEN];// 结果;
9  int main(void)
10  { int i,n;
11  for(i =0;i < DATALEN; + +i)
12  { y[i] =0;
13  for(n =0;n < FILTERTAPS; + +n)
14  if((i - n) > =0)y[i] + =h[n] * x[i - n];
15  }
16  for(i =0;i < DATALEN; + +i)printf("%.2f ",y[i]);
17  return 0;
18  }
```

至少完成如下优化：

1）移走最内循环的 if 语句（第 14 行）；

2）进行循环展开（第 13 行）；

3）进行常数传用；

4）浮点数到定点数的转换；

5）避免对数组的所有访问。

请提供程序进行优化转换后的版本并对结果进行一致性检测。

2. 假设变量 $\{a, b, c, d, e, f\}$ 的访问顺序为（$c\ a\ e\ d\ f\ a\ d\ a\ d\ e\ c\ b\ f\ d\ e\ d\ f\ b\ a\ d\ a$）。

此外，假设处理器有如下特征：

1）只有一个地址寄存器 AR；

2）所有对存储器的访问都必须通过 AR；

3）后增以及后减 1 的操作可以被编码如加载—存储指令中；

4）对 AR 内容进行改变需要额外的指令以及额外的周期。

使用 Liao 的算法，计算一个能够减少总的地址计算时间的可用访问顺序。其中需要将该算法的每个步骤用图形明确表示。

使用汇编语言进行编程，形成一个可用的访问序列。所有的访问都假设是对内存的读（非写）访问。使用如下的汇编指令（具体的指令语义说明在右侧）：

```
ld r,(AR);            register[r]:= memory[AR]
ld r,(AR)++;          register[r]:= memory[AR]; AR++;
ld r,(AR)- -;         register[r]:= memory[AR]; AR- -;
li AR,constant;       AR:=constant;
addi AR,constant;     AR:=AR+constant; //常量可以为负
```

3. 假设计算机配备有主存以及 SPM，存储器的容量以及每次访问所需的功耗如图 7.26 所示。

| 存储器 | 大小/B | 每次访问所需功耗 |
|---|---|---|
| 便签式存储器 | 4096 (4k) | 1.3 nJ |
| 主储存器 | 262,144 (256 k) | 31 nJ |

图 7.26　存储器特征

假设访问如图 7.27 所示的变量。

倘若使用静态的、非覆盖式变量，其中哪些变量应该分配在 SPM 中？使用 ILP 模型选择对应的变量，答案应包括 ILP 模型以及其对应的结论。可以使用 lp_solve 程序［Anonymous，2010a］解决 ILP 问题。

| 变量 | 大小/B | 访问次数 |
|---|---|---|
| $a$ | 1024 | 16 |
| $b$ | 2048 | 1024 |
| $c$ | 512 | 2048 |
| $d$ | 256 | 512 |
| $e$ | 128 | 256 |
| $f$ | 1024 | 512 |
| $g$ | 512 | 64 |
| $h$ | 256 | 512 |

图 7.27　变量特征

4. 循环展开是一种非常有用的优化方式，请指出这种优化方式的两种优点以及两种缺点。

# 第8章　测　　试

## 8.1　总览

测试的目的是保证所制造的嵌入式系统的行为与预期相符。测试既可以在产品生产过程中进行，还可以在产品生产完毕后进行（制造测试），还可以在系统交付给客户后进行（现场测试）。在进行嵌入式测试时，应特别注意如下因素：

1）用于实际环境中的嵌入式物理子系统有可能是与实际安全息息相关的。因此，嵌入式系统与常用的办公设备相比，一旦出现故障则会导致产生更加严重的后果。因此，产品的质量就必须远远高于与安全无关的系统。

2）测试对时间要求严格的系统时必须确保系统的时间行为是正确的。这意味着仅仅对系统功能进行测试是不够的。

3）在嵌入式系统实际的运行环境中进行测试有可能是比较危险的。例如，测试核电站的控制软件可能会导致对测试者产生某些严重的问题。

在设计阶段就应该进行测试的准备工作，这样在产品研发的初期就可以对测试工作提供一定的支持。在设计过程中就可以对产品的可测试性进行设计评估。为了简化第5章的内容，将所有与测试有关的内容放到本章进行说明。尽管在进行实际的产品设计时，测试应尽可能早参与到设计阶段，但是在示例图中，测试放到了设计流程中的最后一步（见图8.1）。然而，将测试放入产品设计阶段并不是一种常见的开发方式，因此图8.1也是与实际的设计流程相对应的。

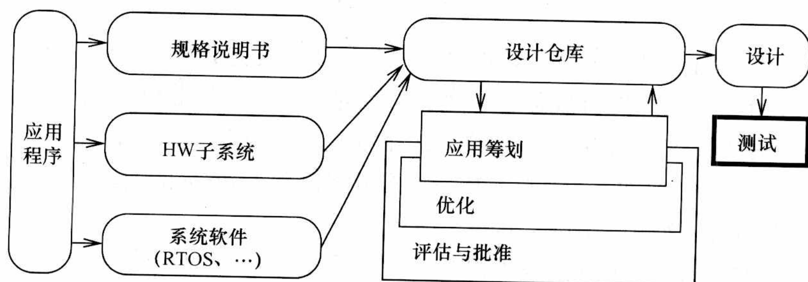



图8.1　测试是设计流程的最后一步

在测试中，通常认为所设计的系统（System Under Design，SUD）就是**被测设备**（Device Under Test，DUT）。在DUT中，会使用一系列特殊的、被选定的被称作测试用例的输入用例输入至系统中，并观察系统行为是否与所预期的行为相符。

这些测试用例通常适用于实际的、已经制造并运行的系统。测试的主要目的是鉴定尚未正确生产的系统以及以生产的系统是否有潜在故障。

测试包括如下步骤：

1）测试用例生成；

2）测试用例的使用；

3）响应观测；

4）结果比较。

## 8.2　测试过程

### 8.2.1　门级别测试用例生成

在测试用例的生成过程中，需要鉴别该组测试用例能否鉴别系统是不是正常运行的。测试用例的生成通常基于**故障模型**，这些故障模型模拟可能的故障。所产生的测试用例尝试生成能够对该错误模型进行所有故障测试的测试用例。

困在故障模型通常用于故障模型中。该模型基于电路中的内部线路永远地被连至'0'或'1'上的假设。据观察，许多实际的故障电路中，其内部线路就被通过该种方式进行了永久性的链接。例如，参考图8.2[㊀]所示的电路。

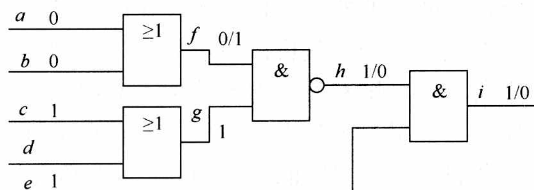



图8.2　门级别的测试用例

假设要检测当信号$f$的固定1故障。为此目的，通过设置$a=b=$'0'让$f$等于0。如果$f$等于1，则此处有故障，否则$f$应当为0。为了观测$f$的值，将通过信号$i$输出结果。为实现这一目标，必须将$e$设置为1并且将$c$或$d$也设置为1。如果没有故障发生，$h$与$i$则为1，否则为0。测试用例包含了$a \sim e$的所有可输入的值。D算法可以用于产生该测试用例［Lala，1985］。

许多用于生成测试用例的技术都是基于固定故障模型的，然而COMS技术需要更多的综合故障模型。在CMOS技术中，故障可以改变设备的内部状态。一旦线路发生损坏（例如stuck-at-open故障），该错误就会发生，因此门电路的晶体管就会

㊀　请记住：为了符合ANSI/IEEE 91标准，符号≥1以及&分别表示或门以及与门。

断开。这些晶体管是否导通，基于线路损坏前的门电路存储的栅极的电荷。这样通过存储的电荷，在栅极就可以“记住”输入的信号。此外，有些故障也有可能由瞬时故障或延迟故障所产生（故障有可能会改变电路的延时）。延迟故障也有可能是由于线路之间相邻导线的信号串扰所导致，故障模型也应将此类硬件故障考虑在内［Krstić and Cheng，1998］。

虽然存在着良好的硬件测试模型，但是该模型并不能用于软件测试。

### 8.2.2　自测程序

对现代集成电路测试的另一个关键问题是其有限的引脚数量使得对内部组件的访问越来越困难。另外，在这些电路全速运行时对其进行测试也非常困难，因此测试者使用的工具必须至少与要测试的电路一样快。事实上，许多嵌入式系统的处理器提供了一种走出该困境的方法：处理器可以运行测试程序或对电路进行诊断。这些诊断的方法用于对主框架机的测试已经有几十年的历史。图8.3显示了有可能被包含在某些处理器中的组件。

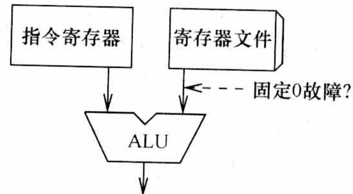



图8.3　处理器硬件分割

为了测试ALU的stuck-at-faults输入错误，可以执行如下一段测试用例：

将所有为’1’的类型存储至寄存器文件中；

将常量“0000...000”与寄存器的值进行异或，

观察其中是否包含’0’bit，

如果结果为真，报告错误；

否则继续进行测试

可以生成类似的小测试用例用于其他故障。不幸的是，处理器为main frames生成的诊断程序几乎都是手动生成的。也有一些研究者致力于自动诊断程序的生成工作［Brahme and Abraham，1984］，［Krüger，1986］，［Bieker and Marwedel，1995］，［Krstic and Dey，2002］，［Kranitis et al.，2003］，［Bernardi et al.，2005］。

## 8.3　测试模式集的评估以及系统的鲁棒性

### 8.3.1　故障覆盖率

测试模式集质量的评估可以以故障覆盖率作为度量标准。故障覆盖率是从给定的测试集中包含的潜在故障的百分比。

故障覆盖率=给定的测试集中可检测的故障数量/故障模型中总故障数量

实际上，若想生产出符合质量要求的产品，需要保证故障覆盖率应能达到至少

98%～99%。对某些特定系统，需要更高的覆盖率。因此，对于某些特定的硬件组件，就需要特定的故障模型（例如电池）。

除了达到高覆盖率外，也必须保证高**正确覆盖率**，这意味着无故障的系统必须达到上述两种要求，否则必须保证系统的故障覆盖率达到100%。

为了增加系统验证时可用的选项数量，应该在系统设计阶段就提出具体的测试方法。例如，可以在系统的软件模型中使用测试用例集，用于检测两种软件模型的行为是否一致。系统如果使用形式化的验证方法用于此种案例中将耗费更长的时间。

### 8.3.2　故障仿真

完全预测系统现存的故障或通过分析计算覆盖率是几乎无法完成的（可以预见的未来也无法完成）。因此，对系统当前故障行为的预测，通常通过仿真进行。这种仿真的方式称作故障仿真。在进行故障仿真时，系统模型应修改为反映系统当前存在的某种故障的行为。

故障仿真的目标包括：

1）了解该组件的故障在系统级别的影响。如果故障不影响系统的可观察行为，我们就称该故障是冗余的。

2）了解提高容错的机制是否对系统有任何帮助。

故障仿真需要对系统中所有故障模型进行可能的故障仿真，并且可能使用大量的、不同的测试用例用于输入。因此，故障仿真是一个非常耗时的过程。通过使用不同的技术可以提高故障仿真的效率。

有一种类似技术用于进行门级别的故障仿真。在这种情况下，内部的信号都是单一的比特信号。这样就可以将信号映射至仿真的宿主机某一机器字的某个比特中。与和或的机器指令也可用于模拟布尔网络，然而每个机器字中可能只有一个比特位使用，这意味着要提高并发故障仿真的效率。在并发故障仿真中，如果 $n$ 等于机器字的长度，同一时刻有 $n$ 个不同测试用例用于仿真。$n$ 个测试用例被映射至该机器字的不同比特中，执行一组相同的与、或指令将会模拟有 $n$ 个测试用例的布尔网络的行为。

### 8.3.3　故障输入

对于实际的物理系统来说，进行故障仿真需要耗费大量的时间。如果有可用的实际系统，就可以使用故障输入替代故障仿真。故障输入不依赖于故障模型（尽管可以使用这些模型），因此故障的输入有可能在不能进行故障预测的故障模型中导致故障的产生。

可以区分如下两种类型的故障输入：

1）系统的局部故障。

2）环境故障（与系统规格需求不相符的行为）。例如，可以检测当指定的温度或辐射强度超出范围时的系统行为。

可以用于故障输入的若干模型如下：

1）在硬件级别的故障输入：示例包括硬件引脚操作、电磁以及核辐射。

2）在软件级别的故障输入：例如对某些内存比特位的触发。

故障输入的质量基于“探针效果”：探针有可能会影响到系统的行为。这些影响应当越小越好，本质上应该是可以忽略的。

根据 Kopetz［Kopetz，1997］的实验报告，基于软件的故障输入本质上与基于硬件的故障输入一样有效。但核辐射是一个明显的例外，因为生成此类错误的方法与生成其他错误的方法有着本质的不同。

## 8.4　可测试性设计

### 8.4.1　动机

在 8.2.1 节已经提出了设计逻辑电路测试用例的思路。对于电路实现状态机而言（自动机），测试用例的生成会更加困难。检测两个有限状态机模型是否等价，需要非常复杂的输入序列［Kohavi，1987］。例如，考虑图 2.27 所示的状态表，为了方便起见，再次在图 8.4 中复现。

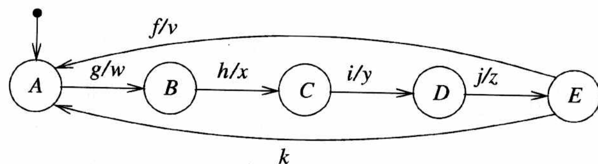



图 8.4　用于测试的有限状态机

假设要测试状态 $C$ 到状态 $D$ 的过渡，通过执行适当的输入序列，首先获得状态 $C$。接下来，必须生成输入事件 $i$，如果生成了输出 $y$，则需要对其进行检测。另外，还需要检测是否达到了状态 $D$。这个过程相当复杂，并且容易受到其他错误的影响㊀。

该示例证明：如果只对系统输出进行测试，那么对系统的测试将会变得非常困难。为了简化测试，可以添加特殊的硬件使得测试变得更加简单。用于简化测试的设计过程，成为可测试性设计（Desigh for Testability，DfT）。用于测试有限状态机的特殊硬件就是一个典型的示例。

㊀ 有限状态机测试的简化原因是有限状态机中包含了对线性链的转换（在本章的思考题中有相关内容）。

## 8.4.2　扫描设计

通过使用扫描设计，可以简化将用例输入程序，达到某一特定状态，并对该状态结果进行观测。在扫描设计时，可以将触发器中存储的状态连接至一个串行移位寄存器中（见图8.5）。

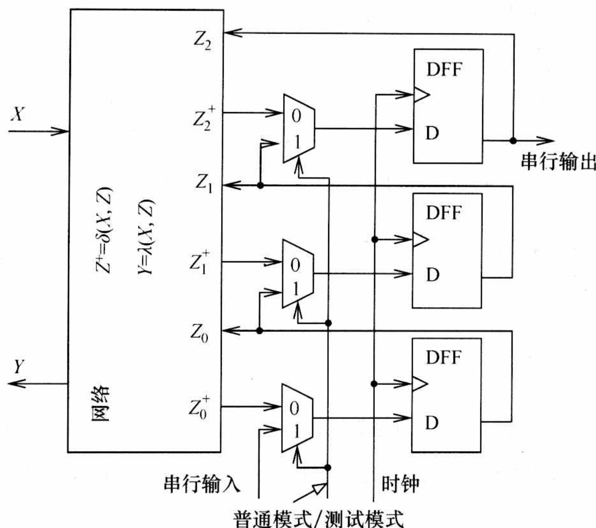



图8.5　扫描路径设计

电路中包含3个D触发器，并且在每个触发器的输入端都包含了一个数据选择器。通过控制该数据选择器的输入（在最底部显示了数据选择器的输入），既可以连接至基于当前输入和当前状态所生成的下一个状态的触发器网络，也可以将触发器连接至一个串行链中。通过将数据选择器设置为扫描模式，可以在状态位进入扫描链后，再载入该状态位（每个时钟周期1bit）。

这样，就可以将任意的状态位顺序下载至3个触发器中。在第二个阶段，当数据选择器处于普通模式时，可以在有限状态机中使用一个输入用例。在下一个时钟滴答，有限状态机将转变状态。新的状态在第三及最终阶段会被连续移出，并且会再次使用串行模式（每个时钟1bit）。人们并不关心如何达到某一状态以及如何观察布尔函数$\delta$的下一个状态是否正确地实现了对有限状态机的测试。实际上，处理的基于状态的系统所影响的只有两个（简单的）阶段，以及无状态的测试用例。布尔网络可以用于检测输出的正确性。

扫描设计技术在单芯片的场景下运行良好。对于板级集成来说，需要某些技术连接若干个芯片的扫描链。JTAG是用于处理该场景的标准。JTAG标准定义了所有芯片的边界寄存器以及测试引脚的数量，另外还定义了将所有芯片连接至扫描链的控制命令。JTAG也被称作边界扫描［Parker，1992］。

### 8.4.3 特征分析

为了避免将 DUT 的响应移出，可以将响应进行压缩。可以使用图 8.6 所示的计划：

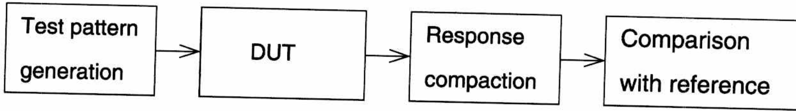



图 8.6 对 DUT 进行测试

生成的测试用例用于 DUT 的输入（或称为激励）。DUT 的响应被压缩形成为一个代表响应的信号。该响应会与预期的响应进行对比，可以通过仿真计算得出预期响应。

压缩通常通过线性反馈移位寄存器（Linear Feedback Shift Registers，LFSR）运行，该移位寄存器使用异或反馈。图 8.7 显示了一个 4bit 的 LFSR（左图）以及其相关的状态图（右图）[Lala，1985]。

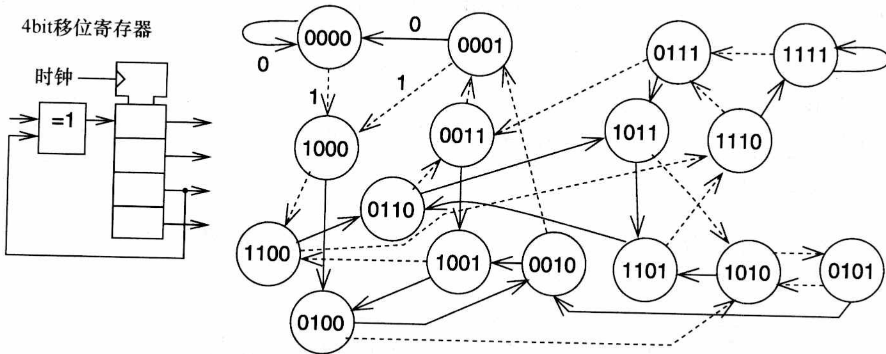



图 8.7 用于响应压缩的线性反馈寄存器

虚线表示输入为’1’，实线表示输入为’0’。选择的反馈产生所有可能的特征。

在测试时，系统测试的响应被发送至 LFSR 的输入中。LFSR 接下来会生成反映响应的信号。由于使用存储的信号替代所有的响应，有些响应的类型可以被映射至同样的信号中。那么从不正确的响应中获取正确的信号的概率会是多少？

总之，一个有 $n$bit 的信号发生器可以生成 $2^n$ 个信号。对于 DUT 的 $m$bit 的响应，可以将 $2^{(m-n)}$ 个响应均匀映射至相同的信号中。假设预期的某一信号用于系统的正确响应，接下来，$2^{(m-n)}-1$ 个不正确的响应也会映射至相同的信号中。如果响应有 $m$bit，那么不正确的响应总数为 $2^m-1$。因此，将不正确的响应映射至正确

的信号（提供到信号的均匀模式映射）中的概率为

$$P = \Pr\left(\frac{\text{映射至相同信号的模型}}{\text{映射模型的总数}}\right) \tag{8.1}$$

$$= \frac{2^{(m-n)} - 1}{2^m - 1} \tag{8.2}$$

$$\approx \frac{1}{2^n} \tag{8.3}$$

此时 $m >> n$。

这意味着如果移位寄存器足够长，从不正确的测试响应中生成正确的信号的概率是非常小的。

### 8.4.4　伪随机测试模式生成

对于使用大量触发器的芯片，需要大量的时间进行测试用例的移动。为了加快芯片生成测试用例的时间，通常会将生成测试用例的硬件也集成至芯片中。

例如，伪随机模式（依然是由 LFSR 生成）也可用于测试模型中。例如可以将图 8.7 所示电路改为图 8.8 所示。

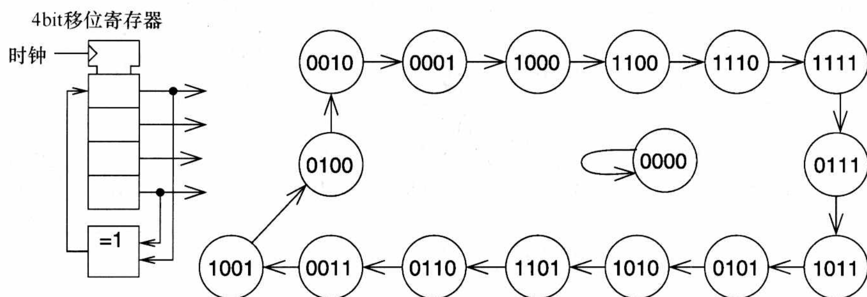



图 8.8　用于生成测试用例的线性反馈移位寄存器

该电路会生成所有除了由全 0 组成用例外的测试用例。由于生成器一旦到达全 0 的状态时会被阻塞，所以必须避开全 0 状态。这种生成的模型在运行被测系统时的效果优于单计数器系统。

### 8.4.5　内置逻辑块观测

内置逻辑块观测（Built-In Logic Block Observer，BILBO）［Könemann et al.，1979］推荐用于电路合并测试用例生成，用于测试密封响应以及连续功能扫描。如图 8.9 所示，每个 BILBO 使用 3 个 D 触发器。

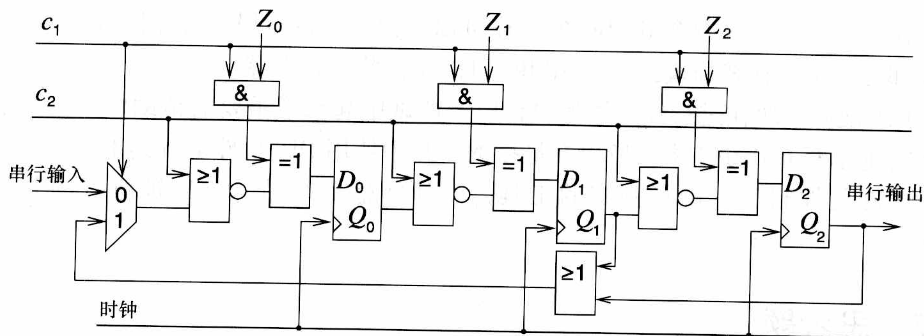



图 8.9　BILBO

图 8.10 显示了 BILBO 的寄存器模式。图 8.9 所示的 3bit 寄存器可以用于通道扫描、复位、线性反馈移位寄存器（Linear-Feedback Shift Register，LFSR）以及普通模式。在 LFSR 模式中，该寄存器可以用于生成伪随机模式或对输入（$Z_0 \sim Z_2$）进行响应压缩。在这种情况下，压缩就会基于并行输入而非之前所提到的串行输入。使用并行输入进行压缩的目的及行为与使用串行输入的目的是一致的。

| $c_1$ | $c_2$ | $D_i$ | |
|---|---|---|---|
| '0' | '0' | $'0' \oplus \overline{Q_{i-1}} = \overline{Q_{i-1}}$ | 扫描路径模式 |
| '0' | '1' | $'0' \oplus \overline{'1'} = '0'$ | 复位 |
| '1' | '0' | $Z_i \oplus \overline{Q_{i-1}}$ | LFSR 模式 |
| '1' | '1' | $Z_i \oplus \overline{'1'} = Z_i$ | 普通模式 |

图 8.10　BILBO 的寄存器模式

特别说明的是，BILBO 通常成对使用（见图 8.11）。

一个 BILBO 会生成一个伪随机测试用例，并将这些用例输入至布尔网络中。布尔网络的响应会被链接到该网络输出的第二个 BILBO 进行压缩。在测试序列的最后，被压缩的响应会被连续移出并且与预期的响应进行对比。预期的相应可以通过仿真进行计算得出。

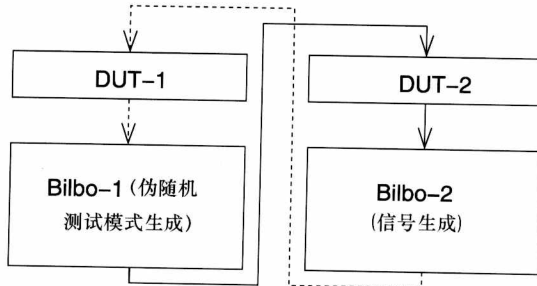



图 8.11　交叉耦合的 BILBO

在第二个阶段，两个 BILBO 的用途可以互换。在该阶段，使用图 8.11 所示的虚线部分表示。在普通模式中，BILBO 可以用作状态寄存器。

DfT 硬件模块在进行硬件原型设计以及调试中起到了重要的帮助作用。由于硬件设备绝不会有 0 缺陷率，所以在最后的产品中使用 DfT 硬件也有其重要意义。使用此类在硬件制造时进行测试的工具，以降低整个产品的支出被所有的公司所青睐。

## 8.5 思考题

1. 考虑如图 8.2 所示的电路，在信号 $h$ 中生成一个故障 0 错误的测试用例。
2. 哪个状态图与图 8.12 所示的 LFSR 相符？
3. 对图 8.4 所示的 FSM，指定其测试用例以及预期的相应。这些用例必须被指定为一系列成对的序列（测试用例、预期响应）。图 8.4 所示的事件可以用作测试用例。假设 FSM 在上电后处于默认状态，提供一个针对所有转移的完整测试！注意，特殊的 FSM 链结构可以简化测试。

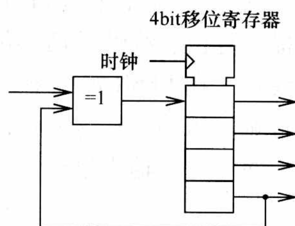


图 8.12 LFSR

# 附　　录

## 附录 A　整数线性规划

整数线性规划（Integer Linear Programming，ILP）是一种数学化的优化技术，它可以用于解决大量的工程优化问题。

ILP 提供了一种比较通用的优化问题建模方式。ILP 模型包含两部分内容：一个成本函数（Cost Function）以及一个约束集（Set of Constraints）。这两部分都会引用到以整数值为变量的集合 $X = \{x_i\}$。成本函数必须是这些变量的线性函数，因此它们可以表示为如下形式：

$$C = \sum_i a_i x_i \tag{A.1}$$

式中　$a_i \in IR$；

$x_i \in IN_0$。

约束集也必须是包含整数变量的线性函数，可以表示为如下形式：

$$\forall j \in J: \sum_i b_{i,j} x_i \geqslant c_j \tag{A.2}$$

式中　$b_{i,j}$、$c_j \in IR$。

**定义**：问题是在式（A.2）给出的约束条件下寻找式（A.1）的极小化成本函数。如果所有的变量被约束为 0 或者 1，则相应的模型被称为 0/1 整数线性规划模型（0/1-Integer Linear Programming Model）。在这种情况下，变量仍然表示的是(二进制）决策变量。

如果约束 $bi$，$j$ 作出相应的修改，则式（A.2）中的≥可以替代为≤。如果在式（A.2）中的非负变量前均乘以 $-1$，则也可以使用负整数 $x_i$（即 $x_i$ 可以是任何整数值)。对于求某些增益函数 $C'$极大值的情况，可以设置 $C = -C'$。

举例，假定 $x_1$、$x_2$ 和 $x_3$ 均为整数，下式表示了一个 0/1IP 模型：

由于约束集的要求，所有变量只能为 0 或 1。在图 A.1 列出了 4 种解法，成本为 9 的解法是最优的。

$$C = 5x_1 + 6x_2 + 4x_3 \tag{A.3}$$

$$x_1 + x_2 + x_3 \geqslant 2 \tag{A.4}$$

$$x_1 \leqslant 1 \tag{A.5}$$

$$x_2 \leqslant 1 \tag{A.6}$$

$$x_3 \leqslant 1 \tag{A.7}$$

ILP是线性规划的一种变异。对于线性规划，其变量可以是任何实数值。使用数学化的方法，ILP与LP模型都可以用于去解决一些优化问题。但不幸的是，ILP是NP完全（NP-Complete）的（LP不是），同时ILP的执行次数可能会非常庞大。

| $x_1$ | $x_2$ | $x_3$ | $C$ |
|---|---|---|---|
| 0 | 1 | 1 | 10 |
| 1 | 0 | 1 | 9 |
| 1 | 1 | 0 | 11 |
| 1 | 1 | 1 | 15 |

图A.1 上述ILP问题的可能解法

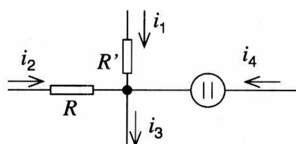
然而，如果模型不是特别庞大，则ILP模型在对优化问题的建模上就非常有效。在对整数线性规划问题进行优化建模时，可以忽略问题的复杂度：许多问题都可以在可接受的时间内解决，如果不能，则ILP模型仍然能作为解决问题的一个起点。ILP的执行时间依赖于变量的数量，以及约束集的数量与结构。一些好的ILP算法（如lp_solve［Anonymous，2010a］或CPLEX）可以在可接受的时间内（如数分钟）解决包含数千个变量的结构良好的问题。关于ILP与LP的更多资料，可以参考某些著作的相关章节（如Wolsey［Wolsey，1998］）。

# 附录B 基尔霍夫定律与运算放大器

在3.6.1节对D-A转换器的讲述中，也涉及了关于运算放大器的基本原理。通常计算机科学专业的学生都比较缺乏对这方面知识的了解，因此在本书的附录B中有必要将这些基本知识再重新阐述一下。由于理解运算放大器的基本知识需要用到基尔霍夫定律（Kirchhoff's Laws），所以也在附录B中对其进行介绍。

## 1. 基尔霍夫定律

基尔霍夫定律是一种电路分析的方法。第一条定律是基尔霍夫电流定律，也称作基尔霍夫节点定律或基尔霍夫第一定律。使用图B.1中的节点来解释这一定律。




图B.1 电路中的节点

**基尔霍夫电流定律**：在电路中的任一节点，流入此节点的电流和与流出此节点的电流和相等［Jewett and Serway，2007］。对于电路中的任一节点有

$$\sum_k i_k = 0 \tag{B.1}$$

按式（B.1）来使用基尔霍夫定律，将流出节点的电流用从节点指向远处的箭头表示，同时计为负值，这种计算的方法其实与电子的真正流向没有关系。

如对于当前的图B.1有

$$i_1 + i_2 - i_3 + i_4 = 0 \tag{B.2}$$

$$i_1 + i_2 + i_4 = i_3 \tag{B.3}$$

由于电荷在转换中的守恒，才产生了这一定律。如果没有这一定律，则电荷数

将不再为常量，而电压也将不断下降。

基尔霍夫的第二个定律应用在电路回路中，也被称为基尔霍夫电压定律或基尔霍夫回路定律，或基尔霍夫第二定律。图 B. 2 展示了一个例子。

**基尔霍夫电压定律**：对于一个闭合回路，在任何时刻沿该回路的电压代数和等于零［Jewett and Serway, 2007］。对于电路中的任一回路有

$$\sum_k V_k = 0 \tag{B.4}$$

如果电压的方向与箭头的方向相反，则将电压计为负值。如对于图 B. 2 中的电路有

$$V_1 - V_2 - V_3 + V_4 = 0 \tag{B.5}$$

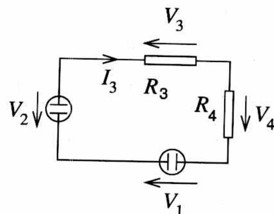



图 B. 2　电路回路

这种不变性的根本原因是能量的转换本质。如果没有这一定律，则在电路回路中增添电荷，也可以不计较能量的损耗。

通常，电子真正流动的方向以及两个端点之间的相对方向并不重要。上述图中的方向可以随意选择，只需要保证在应用基尔霍夫定律时，使用的箭头方向遵从了这一定律即可。如果箭头所表示的加在元器件上的电压与流过元器件的电流方向相反，则在公式中就需要重新考虑此元器件。例如，对于电压与电流方向相反的情况，如果在图 B. 2 中对 $R_3$ 使用欧姆定律（Ohm's Law）：

$$I_3 = -\frac{V_3}{R_3} \tag{B.6}$$

当然，通常都会将电压与电流的方向定义为一致，从而不必过多地去考虑计算中的符号问题。

2. 运算放大器

在某些场合，通常需要将一个信号 $x(t)$ 放大，从而得到另一个当 $a>1$ 时的信号 $y(t)=a\cdot x(t)$，其中 $a$ 被称为**增益**（Gain）。如果真为不同的增益来设计不同的电路，那会是一项非常艰苦的工作，因此工程师通常使用比较容易配置的放大器来得到所需增益。这些放大器即被称为**运算放大器**（Operational Amplifier），或者简写为 op-amp。op-amp 也可以用于非常宽的增益需求范围。在实际中只需要修改电路上 op-amp 周边的一些硬件元器件，就可以得到不同的增益。

通常，运算放大器都有两个信号输入端，一个信号输出端，有两个供电电源（见图 B. 3）。

op-amp 使用增益 $g$，以地为参考来放大两个输入信号之间的电压差：

$$V_{out} = g\cdot(V_+ - V_-) \tag{B.7}$$

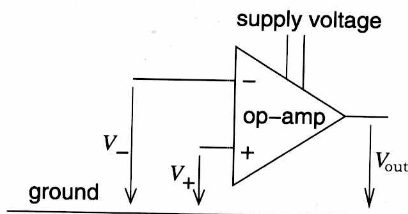



图 B. 3　运算放大器

$g$ 被称为**开环增益**（Open Loop Gain），

通常 $g$ 的值会非常大（$10^4 < g < 10^6$）。对于理想 op-amp，$g$ 可以认为是无穷大的。通常 op-amp 还会有非常大的输入阻抗（$>1\text{M}\Omega$），因此可以忽略信号的输入电流。对于理想运算放大器，其输入阻抗是无穷大的，而输入电流为零。

op-amp 在几十年前就已经商业化了，它们一些是分立的集成电路，另一些被集成到其他电路中。op-amp 的区别往往在于它们的转换速率、电压范围、电流驱动能力以及其他一些特性。op-amp 的实际电路增益通常是通过选择外部电阻来实现的。图 B.4 展示了一个运算放大器的电路。

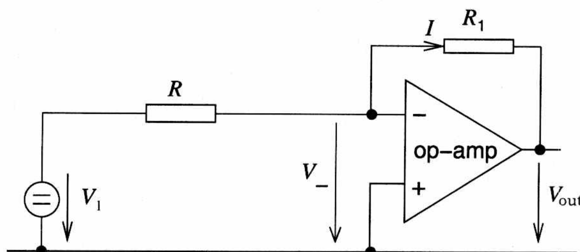



图 B.4 带反馈的运算放大器

两个输入信号之间的任何微小的电压差，都将被乘以一个非常大的放大因子。输出结果电压通过电阻 $R_1$ 反馈到了运算放大器的反向输入端，因此当 $V_-$ 为正电压时，$V_{out}$将为负电压，反之亦然。也就是说因为大的放大率，反馈对输入电压也有了强大的反作用，它会减小输入引脚上的电压。问题在于：减小多少？使用基尔霍夫定律可以计算电压 $V_-$（见图 B.5）。

基于 op-amp 的特性有

$$V_{out} = -g \cdot V_- \tag{B.8}$$

将基尔霍夫定律应用在图 B.5 中所示的回路上有

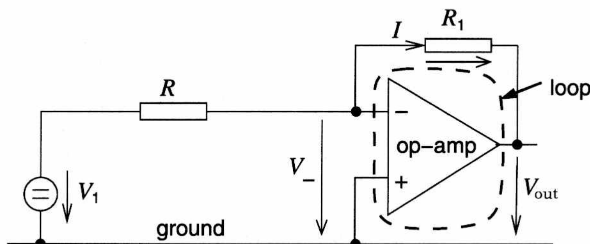



图 B.5 带反馈的运算放大器（被强调的回路）

$$I \cdot R_1 + V_{out} - V_- = 0 \tag{B.9}$$

注意，由于假定的电压方向与箭头方向相反，因此在 $V_-$ 的前面加上了负号。根据式（B.8）与式（B.9）有

$$I \cdot R_1 + (-g) \cdot V_- - V_- = 0 \tag{B.10}$$

$$(1+g)\cdot V_{-}=I\cdot R_1 \tag{B.11}$$

$$V_{-}=\frac{I\cdot R_1}{1+g} \tag{B.12}$$

$$V_{-,\text{ideal}}=\lim_{g\to\infty}\frac{I\cdot R_1}{1+g} \tag{B.13}$$

$$=0 \tag{B.14}$$

这就是说，对于理想 op-amp，$V_{-}$为零。从这一点出发，反向信号输入端也被称为**虚拟地**（Virtual Ground）。但是这一输入端又不能与地直接相连，因为这会改变输入电流。

在第 3 章有一个思考题，就是关于类似图 B.4 中的实际电压增益的计算。

# 参考文献

[Aamodt and Chow, 2000] Aamodt, T. and Chow, P. (2000). Embedded ISA support for enhanced floating-point to fixed-point ANSI C compilation. *3rd ACM Intern. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 128–137.

[Absint, 2002] Absint (2002). Absint: WCET analyses. *http://www.absint.de/wcet.htm*.

[Absint, 2010] Absint (2010). aiT worst-case execution time analyzers. *http://www.absint.de/ait*.

[Accellera Inc., 2003] Accellera Inc. (2003). SystemVerilog 3.1 - Accellera's extensions to Verilog®. *http://www.eda.org/sv/SystemVerilog_3.1_final.pdf*.

[ACM SIGBED, 2010] ACM SIGBED (2010). Home page. *http://www.sigbed.org*.

[ACM/IEEE, 2008] ACM/IEEE (Dec. 2008). Computer science curriculum 2008: An interim revision of CS 2001. *Association for Computing Machinery, IEEE Computer Society, http://www.acm.org/education/curricula/ComputerScience2008.pdf*.

[Ambler, 2003] Ambler, S. (2003). The diagrams of UML 2.0. *http://www.agilemodeling.com/essays/umlDiagrams.htm*.

[Analog Devices Inc. Eng., 2004] Analog Devices Inc. Eng. (2004). *Data Conversion Handbook (Analog Devices)*. Newnes.

[Anonymous, 2010a] Anonymous (2010a). Introduction to lp_solve 5.5.0.14. *http://lpsolve.sourceforge.net*.

[Anonymous, 2010b] Anonymous (2010b). RTJS home page. *http://www.rtsj.org*.

[Araujo and Malik, 1995] Araujo, G. and Malik, S. (1995). Optimal code generation for embedded memory non-homogenous register architectures. *8th Int. Symp. on System Synthesis (ISSS)*, pages 36–41.

[ARM Ltd., 2009a] ARM Ltd. (2009a). AMBA 2 specification. *http://www.arm.com/products/solutions/AMBA_Spec.html*.

[ARM Ltd., 2009b] ARM Ltd. (2009b). Realview compilation tools compiler reference guide. *http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.swdev/index.html*.

[ARTEMIS Joint Undertaking, 2010] ARTEMIS Joint Undertaking (2010). Home page. *https://www.artemis-ju.eu/organisation*.

[Artist Consortium, 2010] Artist Consortium (2010). Home page. *http://www.artist-embedded.org*.

[Atienza et al., 2007] Atienza, D., Baloukas, C., Papadopoulos, L., Poucet, C., Mamagkakis, S., Hidalgo, J. I., Catthoor, F., Soudris, D., and Lanchares, J. (2007). Optimization of dynamic data structures in multimedia embedded systems using evolutionary computation. *In 10th Int. Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 31–40.

[August et al., 1997] August, D. I., Hwu, W. W., and Mahlke, S. (1997). A framework for balancing control flow and predication. *Ann. Workshop on Microprogramming and Microar-*

*chitecture (MICRO)*, pages 92–103.

[AUTOSAR, 2010] AUTOSAR (2010). Automotive open system architecture. *http://www.autosar.org*.

[Avissar et al., 2002] Avissar, O., Barua, R., and Stewart, D. (2002). An optimal memory allocation scheme for scratch-pad-based embedded systems. *Transactions on Embedded Computing Systems*.

[Avižienis et al., 2004] Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

[Azevedo et al., 2002] Azevedo, A., Issenin, I., Cornea, R., Gupta, R., Dutt, N., Veidenbaum, A., and Nicolau, A. (2002). Profile-based dynamic voltage scheduling using program checkpoints. *Design, Automation and Test in Europe (DATE)*, pages 168–175.

[Bäck et al., 1997] Bäck, T., Fogel, D., and Michalewicz, Z. (1997). *Handbook of Evolutionary Computation*. Oxford Univ. Press.

[Bäck and Schwefel, 1993] Bäck, T. and Schwefel, H.-P. (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, pages 1–23.

[Balarin et al., 1998] Balarin, F., Lavagno, L., Murthy, P., and Sangiovanni-Vincentelli, A. (1998). Scheduling for embedded real-time systems. *IEEE Design & Test of Computers*, pages 71–82.

[Ball, 1996] Ball, S. R. (1996). *Embedded Microprocessor Systems - Real world designs*. Newnes.

[Ball, 1998] Ball, S. R. (1998). *Debugging Embedded Microprocessor Systems*. Newnes.

[Banakar et al., 2002] Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., and Marwedel, P. (2002). Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. *10th Intern. Symp. on Hardware/software Codesign (CODES)*, pages 73–78.

[Barney, 2010] Barney, B. (2010). POSIX threads programming. *https://computing.llnl.gov/tutorials/pthreads*.

[Barr, 1999] Barr, M. (1999). *Programming Embedded Systems*. O'Reilly.

[Barrett and Pack, 2005] Barrett, S. and Pack, D. (2005). *Embedded Systems - Design and Applications with the 68HC12 and HCS12*. Prentice Hall.

[Basten, 2008] Basten, T. (2008). Opening remarks, 2nd Artist workshop on models of computation and communication. *Eindhoven, http://www.es.ele.tue.nl/~tbasten/mocc2008/presentations/mocc.pdf*.

[Basu et al., 1999] Basu, A., Leupers, R., and Marwedel, P. (1999). Array index allocation under register constraints in dsp programs. *Int. Conf. on VLSI Design*, pages 330–335.

[Belbachir, 2010] Belbachir, A. N., editor (2010). *Smart cameras*. Springer.

[Bengtsson and Yi, 2004] Bengtsson, J. and Yi, W. (2004). Timed automata: Semantics, algorithms and tools. *In: J. Desel, W. Reisig and G. Rozenberg (eds.): ACPN 2003, Springer LNCS*, 3098:87–124.

[Benini et al., 2000] Benini, L., Bogliolo, A., and De Micheli, G. (2000). A survey of design techniques for system-level dynamic power management. *IEEE Trans. Very Large Scale Integr. Syst.*, 8(3):299–316.

[Benini and De Micheli, 1998] Benini, L. and De Micheli, G. (1998). *Dynamic Power Management – Design Techniques and CAD Tools*. Kluwer Academic Publishers.

[Bergé et al., 1995] Bergé, J.-M., Levia, O., and Rouillard, J. (1995). *High-Level System Modeling*. Kluwer Academic Publishers.

[Bernardi et al., 2005] Bernardi, P., Rebaudengo, M., and Reorda, S. (2005). Using infrastructure IPs to support SW-based self-test of processor cores. *Workshop on Fibres and Optical Passive Components*, pages 22–27.

[Bertolotti, 2006] Bertolotti, I. C. (2006). Real-time embedded operating systems: Standards and perspectives. *In: R. Zurawski (ed.): Embedded Systems Handbook, CRC Press.*

[Beszedes, 2003] Beszedes, A. (2003). Survey of code size reduction methods. *ACM Computing Surveys*, pages 223–267.

[Bieker and Marwedel, 1995] Bieker, U. and Marwedel, P. (1995). Retargetable self-test program generation using constraint logic programming. *32nd annual Design Automation Conference (DAC)*, pages 605–611.

[Bini et al., 2001] Bini, E., Buttazzo, G., and Buttazzo, G. (2001). A hyperbolic bound for the rate monotonic algorithm. *13th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 59–73.

[Boussinot and de Simone, 1991] Boussinot, F. and de Simone, R. (1991). The Esterel language. *Proc. of the IEEE*, Vol. 79, No. 9, pages 1293–1304.

[Bouwmeester et al., 2000] Bouwmeester, D., Ekert, A. and Zeilinger, A. (eds.) (2000). *The Physics of Quantum Information: Quantum Cryptography, Quantum Teleportation, Quantum Computation*. Springer.

[Bouyssounouse and Sifakis, 2005] Bouyssounouse, B. and Sifakis, J., editors (2005). *Embedded Systems Design, The ARTIST Roadmap for Research and Development*. Lecture Notes in Computer Science, Vol. 3436, Springer.

[Brahme and Abraham, 1984] Brahme, D. and Abraham, J. A. (1984). Functional testing of microprocessors. *IEEE Trans. on Computers*, pages 475–485.

[Braun et al., 2010] Braun, A., Bringmann, O., Lettnin, D., and Rosenstiel, W. (2010). Simulation-based verification of the MOST netinterface specification revision 3.0. *Design, Automation and Test in Europe (DATE)*.

[Bremaud, 1999] Bremaud, P. (1999). *Markov Chains*. Springer Verlag.

[Brockmeyer et al., 2003] Brockmeyer, E., Miranda, M., and Catthoor, F. (2003). Layer assignment techniques for low energy in multi-layered memory organisations. *Design, Automation and Test in Europe (DATE)*, pages 1070–1075.

[Broesma, 2004] Broesma, M. (Sep. 2004). Microsoft server crash nearly causes 800-plane pile-up. *Techworld, http://www.techworld.com/opsys/news/index.cfm?newsid=2275.*

[Brooks et al., 2000] Brooks, D., Tiwari, V., and Martonosi, M. (2000). Wattch: a framework for architectural-level power analysis and optimizations. *27th Int. Symp. on Computer Architecture (ISCA)*, pages 83–94.

[Bruno and Bollella, 2009] Bruno, E. and Bollella, G. (2009). *Real-Time Java Programming: With Java RTS*. Prentice Hall.

[Budkowski and Dembinski, 1987] Budkowski, S. and Dembinski, P. (1987). An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3 – 23.

[Burd and Brodersen, 2000] Burd, T. and Brodersen, R. (2000). Design issues for dynamic voltage scaling. *Int. Symp. on Low Power Electronics and Design (ISLPED)*, pages 9–14.

[Burd and Brodersen, 2003] Burd, T. and Brodersen, R. W. (2003). *Energy efficient microprocessor design*. Kluwer Academic Publishers.

[Burkhardt, 2001] Burkhardt, J. (2001). *Pervasive Computing*. Addison-Wesley.

[Burns and Wellings, 1990] Burns, A. and Wellings, A. (1990). *Real-Time Systems and Their Programming Languages*. Addison-Wesley.

[Burns and Wellings, 2001] Burns, A. and Wellings, A. (2001). *Real-Time Systems and Programming Languages (Third Edition)*. Addison Wesley.

[Buttazzo, 2002] Buttazzo, G. (2002). *Hard Real-time computing systems*. Kluwer Academic Publishers, 4th printing.

[Byteflight Consortium, 2003] Byteflight Consortium (2003). Home page. *http://www.byteflight.com*.

[Camposano and Wolf, 1996] Camposano, R. and Wolf, W. (1996). Message from the editors-in-chief. *Design Automation for Embedded Systems*.

[Caspi et al., 2005] Caspi, P., Sangiovanni-Vincentelli, A., Almeida, L., and et al. (2005). Guidelines for a graduate curriculum on embedded software and systems. *ACM Transactions on Embedded Computing Systems (TECS)*, pages 587–611.

[Cederqvist, 2006] Cederqvist, P. (2006). The CVS manual - version management with CVS. *Network Theory Ltd*.

[Ceng et al., 2008] Ceng, J., Castrillón, J., Sheng, W., Scharwächter, H., Leupers, R., Ascheid, G., Meyr, H., Isshiki, T., and Kunieda, H. (2008). MAPS: an integrated framework for MPSoC application parallelization. In *45th annual Design Automation Conference (DAC)*, pages 754–759.

[Chandrakasan et al., 1992] Chandrakasan, A. P., Sheng, S., and Brodersen, R. W. (1992). Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):119–123.

[Chandrakasan et al., 1995] Chandrakasan, A. P., Sheng, S., and Brodersen, R. W. (1995). Low power CMOS digital design. *Kluwer Academic Publishers*.

[Chanet et al., 2007] Chanet, D., Sutter, B. D., Bus, B. D., Put, L. V., and Bosschere, K. D. (2007). Automated reduction of the memory footprint of the linux kernel. *ACM Trans. Embed. Comput. Syst.*, 6(4):23.

[Chen et al., 2006] Chen, G., Ozturk, O., Kandemir, M., and Karakoy, M. (2006). Dynamic scratch-pad memory management for irregular array access patterns. *Design, Automation and Test in Europe (DATE)*, pages 931–936.

[Chen et al., 2007] Chen, K., Sztipanovits, J., and Neema, S. (2007). Compositional specification of behavioral semantics. *Design, Automation and Test in Europe (DATE)*, pages 906–911.

[Chen et al., 2010] Chen, X., Dick, R., and Shang, L. (2010). Properties of and improvements to time-domain dynamic thermal analysis algorithms. *Design, Automation and Test in Europe (DATE)*.

[Chetto et al., 1990] Chetto, H., Silly, M., and Bouchentouf, T. (1990). Dynamic scheduling of real-time tasks under precedence constraints. *Journal of Real-Time Systems, 2*.

[Chung et al., 2001] Chung, E.-Y., Benini, L., and De Micheli, G. (2001). Source code transformation based on software cost analysis. *Int. Symp. on System Synthesis (ISSS)*, pages 153–158.

[Clarke and et al., 2003] Clarke, E. and et al. (2003). Model checking@CMU. *http://www-2.cs.cmu.edu/~modelcheck/index.html.*

[Clarke et al., 2005] Clarke, E. M., Grumberg, O., Hiraishi, H., Jha, S., Long, D. E., McMillan, K. L., and Ness, L. A. (2005). Verification of the futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232.

[Clavier and Gaj, 2009] Clavier, C. and Gaj, K. (2009). Int. workshop on cryptographic hardware and embedded systems (CHES).

[Clouard et al., 2003] Clouard, A., Jain, K., Ghenassia, F., Maillet-Contoz, L., and Strassen, J. (2003). Using transactional models in SoC design flow. *In: [Müller et al., 2003]*, pages 29–64.

[Coelho, 1989] Coelho, D. R. (1989). *The VHDL handbook*. Kluwer Academic Publishers.

[Coello et al., 2007] Coello, C. A. C., Lamont, G. B., and Veldhuizen, D. A. v. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer.

[Collins-Sussman et al., 2008] Collins-Sussman, B., Fitzpatrick, B., and Pilato, C. (2008). Version control with subversion – for subversion 1.5. *http://svnbook.red-bean.com/en/1.5/svn-book.pdf.*

[Cooling, 2003] Cooling, J. (2003). *Software Engineering for Real-Time Systems*. Addison Wesley.

[Cortadella et al., 2000] Cortadella, J., Kondratyev, A., Lavagno, L., Massot, M., Moral, S., Passerone, C., Watanabe, Y., and Sangiovanni-Vincentelli, A. (2000). Task generation and compile-time scheduling for mixed data-control embedded software. *37th Design Automation Conference (DAC)*, pages 489–494.

[Coussy and Morawiec, 2008] Coussy, P. and Morawiec, A. (2008). *High-Level Synthesis*. Springer.

[Craig, 2006] Craig, I. D. (2006). *Virtual Machines*. Springer.

[Damm and Harel, 2001] Damm, W. and Harel, D. (2001). LSCs: Breathing life into message

sequence charts. *Formal Methods in System Design.*

[Dasgupta, 1979] Dasgupta, S. (1979). The organization of microprogram stores. *ACM Computing Surveys*, Vol. 11, pages 39–65.

[Davis et al., 2001] Davis, J., Hylands, C., Janneck, J., Lee, E. A., Liu, J., Liu, X., Neuendorffer, S., Sachs, S., Stewart, M., Vissers, K., Whitaker, P., and Xiong, Y. (2001). Overview of the Ptolemy project. *Technical Memorandum UCB/ERL M01/11; http://ptolemy.eecs.berkeley.edu.*

[De Greef et al., 1997a] De Greef, E., Catthoor, F., and Man, H. (1997a). Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Proc. Workshop on Parallel Processing and Multimedia*, pages 84–98.

[De Greef et al., 1997b] De Greef, E., Catthoor, F., and Man, H. D. (1997b). Array placement for storage size reduction in embedded multimedia systems. *IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 66–75.

[De Micheli et al., 2002] De Micheli, G., Ernst, R., and Wolf, W. (2002). *Readings in Hardware/Software Co-Design.* Academic Press.

[Deutsches Institut für Normung, 1997] Deutsches Institut für Normung (1997). *DIN 66253, Programmiersprache PEARL, Teil 2 PEARL 90.* Beuth-Verlag; English version available through *http://www.din.de.*

[Dibble, 2008] Dibble, P. C. (2008). *Real-Time Java Platform Programming: Second Edition.* BookSurge Publishing.

[Diederichs et al., 2008] Diederichs, C., Margull, U., Slomka, F., and Wirrer, G. (2008). An application-based EDF scheduler for OSEK/VDX. *Design, Automation and Test in Europe (DATE)*, pages 1045–1050.

[Dierickx, 2000] Dierickx, B. (2000). CMOS image sensors - concepts, Photonics West 2000 short course. *http://www.cypress.com/?rID=14527.*

[Dill and Alur, 1994] Dill, D. and Alur, R. (1994). A theory of timed automata. *Theoretical Computer Science*, pages 183–235.

[Dominguez et al., 2005] Dominguez, A., Udayakumaran, S., and Barua, R. (2005). Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 1(4):521–540.

[Donald and Martonosi, 2006] Donald, J. and Martonosi, M. (2006). Techniques for multicore thermal management: Classification and new exploration. *SIGARCH Comput. Archit. News*, 34(2):78–88.

[Douglass, 2000] Douglass, B. P. (2000). *Real-Time UML, 2nd edition.* Addison Wesley.

[Drusinsky and Harel, 1989] Drusinsky, D. and Harel, D. (1989). Using statecharts for hardware description and synthesis. *IEEE Trans. on Computer Design*, pages 798–807.

[Dulong et al., 2001] Dulong, C., Shrivastav, P., and Refah, A. (2001). The making of a compiler for the Intel® Itanium$^{TM}$ processor. *Intel Technology Journal Q3, http://download.intel.com/technology/itj/q32001/pdf/art_4.pdf.*

[Dunn, 2002] Dunn, W. (2002). *Practical Design of Safety-Critical Computer Systems.* Relia-

bility Press.

[Ecker et al., 2009] Ecker, W., Müller, W., and Dömer, R. (2009). *Hardware-dependent software - Principles and practice*. Springer.

[Edwards, 2001] Edwards, S. (2001). Dataflow languages. *http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/presentations/dataflow.ppt*.

[Edwards, 2006] Edwards, S. (2006). Languages for embedded systems. *In: R. Zurawski (ed.): Embedded Systems Handbook, CRC Press*.

[Egger et al., 2006] Egger, B., Lee, J., and Shin, H. (2006). Scratchpad memory management for portable systems with a memory management unit. *9rd ACM Intern. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 321–330.

[Eggermont, 2002] Eggermont, L. (2002). Embedded systems roadmap. *STW, http://www.stw.nl/NR/rdonlyres/3E59AA43-68B1-4E83-BA95-20376EB00560/0/ESRversion1.pdf*.

[Eichenberger et al., 2005] Eichenberger, A. E., O'Brien, K., O'Brien, Kevin, Wu, P. , Chen, T., Oden, P. H., Prener, D. A., Shepherd, J. C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., and Gschwind, M. (2005). Optimizing compiler for a CELL processor. *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 161–172.

[Elsevier B.V., 2010a] Elsevier B.V. (2010a). Sensors and actuators A: Physical. *An International Journal*.

[Elsevier B.V., 2010b] Elsevier B.V. (2010b). Sensors and actuators B: Chemical. *An International Journal*.

[Esterel Technologies, 2010] Esterel Technologies (2010). Scade suite$^{TM}$ - the standard for the development of safety-critical embedded software in aerospace & defense, rail transportation, energy and heavy equipment industries. *http://www.esterel-technologies.com/products/scade-suite*.

[Esterel Technologies Inc., 2010] Esterel Technologies Inc. (2010). Homepage. *http://www.esterel-technologies.com*.

[European Commission Cordis, 2010] European Commission Cordis (2010). Seventh Framework Programme (FP7). *http://cordis.europa.eu/fp7*.

[Evidence, 2010] Evidence (2010). Erika enterprise. *http://erika.tuxfamily.org*.

[Falk, 2009] Falk, H. (2009). WCET-aware register allocation based on graph coloring. *Proceedings of the 46th Design Automation Conference (DAC)*, pages 726–731.

[Falk and Marwedel, 2003] Falk, H. and Marwedel, P. (2003). Control flow driven splitting of loop nests at the source code level. *Design, Automation and Test in Europe (DATE)*, pages 410–415.

[Falk et al., 2006] Falk, H., Wagner, J., and Schaefer, A. (2006). Use of a Bit-true Data Flow Analysis for Processor-Specific Source Code Optimization. In *4th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 133–138, Seoul/Korea.

[Fettweis et al., 1998] Fettweis, G., Weiss, M., Drescher, W., Walther, U., Engel, F., Kobayashi,

S., and Richter, T. (1998). Breaking new grounds over 3000 MMAC/s: a broadband mobile multimedia modem DSP. *Intern. Conf. on Signal Processing Application & Technology (ICSPA)*, available at *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.9340*.

[Fiorin et al., 2007] Fiorin, L., Palermo, G., Lukovic, S., and Silvano, C. (2007). A data protection unit for NoC-based architectures. In *5th IEEE/ACM Int. Conf. on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 167–172.

[Fisher and Dietz, 1998] Fisher, R. and Dietz, H. G. (1998). Compiling for SIMD within a single register. *Annual Workshop on Lang. & Compilers for Parallel Computing (LCPC)*, pages 290–304.

[Fisher and Dietz, 1999] Fisher, R. J. and Dietz, H. G. (1999). The Scc compiler: SWARing at MMX and 3DNow! *Annual Workshop on Lang. & Compilers for Parallel Computing (LCPC)*, pages 399–414.

[FlexRay Consortium, 2002] FlexRay Consortium (2002). Flexray® requirement specification. version 2.01. *http://www.flexray.de*.

[Fowler and Scott, 1998] Fowler, M. and Scott, K. (1998). *UML Distilled - Applying the Standard Object Modeling Language*. Addison-Wesley.

[Franke, 2008] Franke, B. (2008). Fast cycle-approximate instruction set simulation. In *10th Int. Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 69–78.

[Franke and O'Boyle, 2005] Franke, B. and O'Boyle, M. F. (2005). A complete compiler approach to auto-parallelizing C programs for multi-DSP systems. *IEEE Transactions on Parallel and Distributed Systems*, 16:234–245.

[Freescale semiconductor, 2005] Freescale semiconductor (2005). ColdFire® family programmer's reference manual. *http://www.freescale.com/files/dsp/doc/ref_manual/CFPRM.pdf*.

[Fu et al., 1987] Fu, K., Gonzalez, R., and Lee, C. (1987). *Robotics*. McGraw-Hill.

[Gajski and Kuhn, 1983] Gajski, D. and Kuhn, R. (1983). New VLSI tools. *IEEE Computer*, pages 11–14.

[Gajski et al., 1994] Gajski, D., Vahid, F., Narayan, S., and Gong, J. (1994). *Specification and Design of Embedded Systems*. Prentice Hall.

[Gajski et al., 2000] Gajski, D., Zhu, J., Dömer, R., Gerstlauer, A., and Zhao, S. (2000). *SpecC: Specification Language Methodology*. Kluwer Academic Publishers.

[Gajski et al., 2009] Gajski, D. D., Abdi, S., Gerstlauer, A., and Schirner, G. (2009). *Embedded System Design*. Springer, Heidelberg.

[Ganssle, 2008] Ganssle, J., editor (2008). *Embedded Systems (World Class Designs)*. Newnes.

[Ganssle, 2000] Ganssle, J. G. (2000). *The Art of Designing Embedded Systems*. Newnes.

[Ganssle et al., 2008] Ganssle, J. G., Noergaard, T., Eady, F., Edwards, L., Katz, D. J., Gentile, R., Arnold, K., Hyder, K., and Perrin, B. (2008). *Embedded Hardware - Know it all*. Newnes.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability*. Bell Labaratories, Murray Hill, New Jersey.

[Garg and Khatri, 2009] Garg, R. and Khatri, S. (2009). *Analysis and Design of Resilient VLSI Circuits*. Springer.

[Gebotys, 2010] Gebotys, C. (2010). *Security in Embedded Devices*. Springer.

[Geffroy and Motet, 2002] Geffroy, J.-C. and Motet, G. (2002). *Design of Dependable computing Systems*. Kluwer Academic Publishers.

[Gelsen, 2003] Gelsen, O. (2003). Organic displays enter consumer electronics. *Opto & Laser Europe, June; availabe at http://optics.org/cws/article/articles/17598.*

[Gerber et al., 2005] Gerber, R., Bik, A. J. C., Smith, K., and Tian, X. (2005). *The Software Optimization Cookbook Second Edition. High Performance Recipes for IA 32 Platforms.* Intel Press.

[Gomez and Fernandes, 2010] Gomez, L. and Fernandes, J. (2010). *Behavioral Modeling for Embedded Systems and Technologies*. IGI Global.

[Grötker et al., 2002] Grötker, T., Liao, S., and Martin, G. (2002). *System design with SystemC.* Springer.

[Gupta, 2002] Gupta, R. (2002). Tasks and task management. *Course ICS 212, Winter 2002, UC Irvine, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.7.8704&rep=rep1&type=pdf.*

[Ha, 2007] Ha, S. (2007). Model-based programming environment of embedded software for mpsoc. *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 330–335.

[Halbwachs, 1998] Halbwachs, N. (1998). Synchronous programming of reactive systems, a tutorial and commented bibliography. *Tenth International Conference on Computer-Aided Verification, CAV'98, LNCS 1427, Springer Verlag; see also: http://www.springerlink.com/content/5127074271136j71/fulltext.pdf.*

[Halbwachs, 2008] Halbwachs, N. (2008). Personal communication. *South American Artist School on Embedded Systems, Florianopolis.*

[Halbwachs et al., 1991] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous dataflow language LUSTRE. *Proc. of the IEEE Trans. on Software Engineering*, 79:1305–1320.

[Hansmann, 2001] Hansmann, U. (2001). *Pervasive Computing*. Springer Verlag.

[Harbour, 1993] Harbour, M. G. (1993). RT-POSIX: An overview. *http://www.ctr.unican.es/publications/mgh-1993a.pdf.*

[Harel, 1987] Harel, D. (1987). StateCharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274.

[Hattori, 2007] Hattori, T. (2007). MPSoC approaches for low-power embedded SoC's. *Int. Forum on. Application Specific Multi Processor SoC, http://www.mpsoc-forum.org/2007/slides/Hattori.pdf.*

[Haugen and Moller-Pedersen, 2006] Haugen, O. and Moller-Pedersen, B. (2006). Introduction to UML and the modeling of embedded systems. *In: R. Zurawski (ed.): Embedded Systems Handbook, CRC Press.*

[Hayes, 1982] Hayes, J. (1982). A unified switching theory with applications to VLSI design. *Proceedings of the IEEE, Vol. 70*, pages 1140–1151.

[Heath, 2000] Heath, S. (2000). *Embedded System Design*. Newnes.

[Henia et al., 2005] Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., and Ernst, R. (2005). System level performance analysis - the SymTA/S approach. *IEEE Computers and Digital Techniques*, pages 148–166.

[Hennessy and Patterson, 2002] Hennessy, J. L. and Patterson, D. A. (2002). *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers Inc.

[Hennessy and Patterson, 2008] Hennessy, J. L. and Patterson, D. A. (2008). *Computer Organization – The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc.

[Herken, 1995] Herken, R. (1995). *The Universal Turing Machine: A half-century survey*. Springer.

[Herrera et al., 2003a] Herrera, F., Fernández, V., Sánchez, P., and Villar, E. (2003a). Embedded software generation from SystemC for platform based design. *In: [Müller et al., 2003]*, pages 247–272.

[Herrera et al., 2003b] Herrera, F., Posadas, H., Sánchez, P., and Villar, E. (2003b). Systemic embedded software generation from SystemC. *Design, Automation and Test in Europe (DATE)*, pages 10142–10149.

[Hoare, 1985] Hoare, C. (1985). *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science.

[Hopcroft et al., 2006] Hopcroft, J., Motwani, R., and Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley.

[Horn, 1974] Horn, W. (1974). Some simple scheduling algorithms. *Naval Research Logistics Quarterly, Vol. 21*, pages 177–185.

[Huang and Xu, 2010] Huang, L. and Xu, Q. (2010). AgeSim: A simulation framework for evaluating the lifetime reliability of processor-based SoCs. *Design, Automation and Test in Europe (DATE)*.

[Huerlimann, 2003] Huerlimann, D. (2003). Opentrack home page. *http://www.opentrack.ch*.

[Hüls, 2002] Hüls, T. (2002). Optimizing the energy consumption of an MPEG application (in German). *Master thesis, CS Dept., Univ. Dortmund, http://ls12-www.cs.uni-dortmund.de/publications/theses*.

[Hunt et al., 2007] Hunt, V. D., Puglia, A., and Puglia, M. (2007). *RFID: a guide to radio frequency identification*. Wiley.

[IBM, 2002] IBM (2002). Security: User authentication. *http://www.pc.ibm.com/us/security/userauth.html*.

[IBM, 2009] IBM (2009). What's new in Rational Rhapsody 7.5.1. *http://www.ibm.com/developerworks/rational/library/09/whatsnewinrationalrhapsody-7-5-1*.

[IBM, 2010a] IBM (2010a). IBM Rational StateMate. *http://www.ibm.com/developerworks/*

*rational/products/statemate/.*

[IBM, 2010b] IBM (2010b). Rational DOORS. *http://www-01.ibm.com/software/awdtools/doors/.*

[ICD Staff, 2010] ICD Staff (2010). ICD-C compiler framework. *http://www.icd.de/es/icd-c.*

[IEC, 2002] IEC (2002). IEC 60848 – GRAFCET specification language for sequential function charts. *http://webstore.iec.ch/preview/info_iec60848{ed2.0}b.pdf.*

[IEEE, 1991] IEEE (1991). IEEE graphic symbols for logic functions std 91a-1991. *http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=27895.*

[IEEE, 1997] IEEE (1997). *IEEE Standard VHDL Language Reference Manual (1076-1997).* IEEE.

[IEEE, 2002] IEEE (2002). *IEEE Standard VHDL Language Reference Manual (1076-2002).* IEEE.

[IEEE, 2009] IEEE (2009). IEEE Standard for SystemVerilog- unified hardware design, specification, and verification language. *http://www.ieee.org.*

[IMEC, 1997] IMEC (1997). LIC-SMARTpen identifies signer. *IMEC Newsletter, http://www2.imec.be/content/user/File/Newsletters/newsletter_18.pdf.*

[IMEC, 2010] IMEC (2010). ADRES multimedia processor & 3mf multimedia platform. *http://www2.imec.be/content/user/File/ADRES_3MF.pdf.*

[Intel, 2004] Intel (2004). Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor - White paper. *ftp://download.intel.com/design/network/papers/30117401.pdf.*

[Intel, 2008] Intel (2008). Motion estimation with Intel® streaming SIMD extensions 4 (Intel® SSE4). *http://software.intel.com/en-us/articles/motion-estimation-with-intel-streaming-simd-extensions-4-intel-sse4.*

[Intel, 2010a] Intel (2010a). Intel® AVX. *http://software.intel.com/en-us/avx.*

[Intel, 2010b] Intel (2010b). Intel Itanium processor family. *http://www.intel.com/itcenter/products/itanium.*

[Ishihara and Yasuura, 1998] Ishihara, T. and Yasuura, H. (1998). Voltage scheduling problem for dynamically variable voltage processors. *Intern. Symp. on Low Power Electronics and Design (ISLPED)*, pages 197–202.

[Israr and Huss, 2008] Israr, A. and Huss, S. (2008). Specification and design considerations for reliable embedded systems. *Design, Automation and Test in Europe (DATE)*, pages 1111–1116.

[IT Facts, 2010] IT Facts (2010). Home page. *http://www.itfacts.biz.*

[ITRS Organization, 2009] ITRS Organization (2009). International technology roadmap for semiconductors (ITRS). *http://public.itrs.net.*

[Iyer and Marculescu, 2002] Iyer, A. and Marculescu, D. (2002). Power and performance evaluation of globally asynchronous locally synchronous processors. *Int. Symp. on Computer*

*Architecture (ISCA)*, pages 158–168.

[Jackson, 1955] Jackson, J. (1955). Scheduling a production line to minimize maximum tardiness. *Management Science Research Project 43, University of California, Los Angeles.*

[Jackson et al., 2009] Jackson, J., Marwedel, P., and Ricks, K. (2009). Workshop on embedded system education. *http://www.artist-embedded.org/artist/WESE-09.html.*

[Jacome et al., 2000] Jacome, M., de Veciana, G., and Lapinskii, V. (2000). Exploring performance tradeoffs for clustered VLIW ASIPs. *IEEE Int. Conf. on Computer-Aided Design (ICCAD)*, pages 504–510.

[Jacome and de Veciana, 1999] Jacome, M. F. and de Veciana, G. (1999). Lower bound on latency for VLIW ASIP datapaths. *IEEE Int. Conf. on Computer-Aided Design (ICCAD)*, pages 261–269.

[Jain et al., 2001] Jain, M., Balakrishnan, M., and Kumar, A. (2001). ASIP design methodologies: Survey and issues. *14th Int. Conf. on VLSI Design*, pages 76–81.

[Janka, 2002] Janka, R. (2002). *Specification and Design Methodology for Real-Time Embedded Systems.* Kluwer Academic Publishers.

[Jantsch, 2004] Jantsch, A. (2004). *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation.* Morgan Kaufmann.

[Jantsch, 2006] Jantsch, A. (2006). Models of embedded computation. *In: R. Zurawski (ed.): Embedded Systems Handbook, CRC Press.*

[Java Community Process, 2002] Java Community Process (2002). JSR-1 – real-time specification for Java. *http://www.jcp.org/en/jsr/detail?id=1.*

[Jewett and Serway, 2007] Jewett, J. W. and Serway, R. A. (2007). *Physics for scientists and engineers with modern physics.* Thomson Higher Education.

[Jha and Dutt, 1993] Jha, P. and Dutt, N. (1993). Rapid estimation for parameterized components in high-level synthesis. *IEEE Transactions on VLSI Systems*, pages 296–303.

[Johnson, 2010] Johnson, S. C. (2010). The Lex & Yacc Page. *http://dinosaur.compilertools.net.*

[Jones, 1997] Jones, M. (1997). What really happened on Mars Rover Pathfinder. *In: P.G. Neumann (ed.): comp.risks, The Risks Digest, Vol. 19, Issue 49; available at http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/Mars_Pathfinder.html.*

[Jones, 1996] Jones, N. D. (1996). An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503.

[JXTA Community, 2010] JXTA Community (2010). Home page. *https://jxta.dev.java.net.*

[Kahn, 1974] Kahn, G. (1974). The semantics of a simple language for parallel programming. *Proc. of the Int. Federation for Information Processing (IFIP)*, pages 471–475.

[Kandemir et al., 2001] Kandemir, M., Ramanujam, J., Irwin, M. J., Vijaykrishnan, N., Kadayif, I., and Parikh, A. (2001). Dynamic management of scratch-pad memory space. *38th annual Design Automation Conference (DAC)*, pages 690–695.

[Karp and Miller, 1966] Karp, R. M. and Miller, R. E. (1966). Properties of a model for parallel computations: Determinancy, termination, queueing. *SIAM Journal of Applied Mathematics*, 14:1390–1411.

[Keding et al., 1998] Keding, H., Willems, M., Coors, M., and Meyr, H. (1998). FRIDGE: A fixed-point design and simulation environment. *Design, Automation and Test in Europe (DATE)*, pages 429–435.

[Keinert et al., 2009] Keinert, J., Streubühr, M., Schlichter, T., Falk, J., Gladigau, J., Haubelt, C., Teich, J., and Meredith, M. (2009). SystemCodesigner - an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Transactions on Design Automation of Electronic Systems*, 14:1–23.

[Kempe, 1995] Kempe, M. (1995). Ada 95 reference manual, ISO/IEC standard 8652:1995. *(HTML-version), http://www.adahome.com/rm95/.*

[Kempe Software Capital Enterprises (KSCE), 2010] Kempe Software Capital Enterprises (KSCE) (2010). Ada home: The web site for Ada. *http://www.adahome.com.*

[Kernighan and Ritchie, 1988] Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall.

[Kienhuis et al., 2000] Kienhuis, B., Rijjpkema, E., and Deprettere, E. (2000). Compaan: Deriving process networks from Matlab for embedded signal processing architectures. *Proc. 8th Int. Workshop on Hardware/Software Codesign (CODES)*, pages 29–40.

[Klaiber, 2000] Klaiber, A. (2000). The technology behind Crusoe$^{TM}$ processors. *http://web.archive.org/web/20010602205826/www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf.*

[Könemann et al., 1979] Könemann, B., Mucha, J., and Zwiehoff, G. (1979). Built-in logic block observer. *IEEE Int. Test Conf.*, pages 261–266.

[Ko and Koo, 1996] Ko, M. and Koo, I. (1996). An overview of interactive video on demand system. *www.ece.ubc.ca/~irenek/techpaps/vod/vod.html.*

[Kobryn, 2001] Kobryn, C. (2001). UML 2001: A standardization Odyssey. *Communications of the ACM (CACM), available at http://www.omg.org/attachments/pdf/UML_2001_CACM_Oct99_p29-Kobryn.pdf*, pages 29–36.

[Kohavi, 1987] Kohavi, Z. (1987). *Switching and Finite Automata Theory*. Tata McGraw-Hill Publishing Company, New Delhi, 9th reprint.

[Koninklijke Philips Electronics N.V., 2003] Koninklijke Philips Electronics N.V. (2003). Ambient intelligence. *http://www.research.philips.com/technologies/projects/ambintel.html.*

[Koopman and Upender, 1995] Koopman, P. J. and Upender, B. P. (1995). Time division multiple access without a bus master. *United Technologies Research Center, UTRC Technical Report RR-9500470, http://www.ece.cmu.edu/~koopman/jtdma/jtdma.html.*

[Kopetz, 1997] Kopetz, H. (1997). *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers.

[Kopetz, 2003] Kopetz, H. (2003). Architecture of safety-critical distributed real-time systems. *Invited Talk; Design, Automation and Test in Europe (DATE).*

[Kopetz and Grunsteidl, 1994] Kopetz, H. and Grunsteidl, G. (1994). TTP –a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27:14–23.

[Krall, 2000] Krall, A. (2000). Compilation techniques for multimedia extensions. *International Journal of Parallel Programming*, 28:347–361.

[Kranitis et al., 2003] Kranitis, N., Paschalis, A., Gizopoulos, D., and Zorian, Y. (2003). Instruction-based self-testing of processor cores. *Journal of Electronic Testing*, 19:103–112.

[Krhovjak and Matyas, 2006] Krhovjak, J. and Matyas, V. (2006). Secure hardware - pv018. *http://www.fi.muni.cz/~xkrhovj/lectures/2006_PV018_Secure_Hardware_slides.pdf*.

[Krishna and Shin, 1997] Krishna, C. and Shin, K. G. (1997). *Real-Time Systems*. McGraw-Hill, Computer Science Series.

[Krstić and Cheng, 1998] Krstić, A. and Cheng, K. (1998). *Delay fault testing of VLSI circuits*. Kluwer Academic Publishers.

[Krstic and Dey, 2002] Krstic, A. and Dey, S. (2002). Embedded software-based self-test for programmable core-based designs. *IEEE Design & Test*, pages 18–27.

[Krüger, 1986] Krüger, G. (1986). Automatic generation of self-test programs: A new feature of the MIMOLA design system. *23rd annual Design Automation Conference (DAC)*, pages 378–384.

[Kuchcinski, 2002] Kuchcinski, K. (2002). System partitioning (course notes). *http://www.cs.lth.se/home/Krzysztof_Kuchcinski/DES/Lectures/Lecture7.pdf*.

[Kwok and Ahmad, 1999] Kwok, Y.-K. and Ahmad, I. (1999). Static scheduling algorithms for allocation directed task graphs to multiprocessors. *ACM Computing Surveys*, 31:406–471.

[Labrosse, 2000] Labrosse, J. (2000). *Embedded Systems Building Blocks - Complete and Ready-to-use Modules in C*. Elsevier.

[Lala, 1985] Lala, P. (1985). *Fault tolerant and Fault Testable Hardware Design*. Prentice Hall.

[Lam et al., 1991] Lam, M. S., Rothberg, E. E., and Wolf, M. E. (1991). The cache performance and optimizations of blocked algorithms. *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74.

[Landwehr and Marwedel, 1997] Landwehr, B. and Marwedel, P. (1997). A new optimization technique for improving resource exploitation and critical path minimization. *10th Int. Symp. on System Synthesis (ISSS)*, pages 65–72.

[Lapinskii et al., 2001] Lapinskii, V., Jacome, M. F., and de Veciana, G. (2001). Application-specific clustered VLIW datapaths: Early exploration on a parameterized design space. Technical Report UT-CERC-TR-MFJ/GDV-01-1, Computer Engineering Research Center, University of Texas at Austin.

[Laplante, 1997] Laplante, P. (1997). *Real-Time Systems: Design and Analysis - An Engineer's Handbook*. IEEE Press.

[Laprie, 1992] Laprie, J. C., editor (1992). *Dependability: basic concepts and terminology in English, French, German, Italian and Japanese*. IFIP WG 10.4, Dependable Computing and Fault Tolerance, In: volume 5 of Dependable computing and fault tolerant systems, Springer Verlag.

[Larsen and Amarasinghe, 2000] Larsen, S. and Amarasinghe, S. (2000). Exploiting superword parallelism with multimedia instructions sets. *Programming Language Design and Implementation (PLDI)*, pages 145–156.

[Latendresse, 2004] Latendresse, M. (2004). The code compression bibliography. *http://www.iro.umontreal.ca/~latendre/compactBib*.

[Law, 2006] Law, A. M. (2006). *Simulation Modeling & Analysis*. McGraw-Hill.

[Lawler, 1973] Lawler, E. L. (1973). Optimal sequencing of a single machine subject to precedence constraints. *Managements Science, Vol. 19*, pages 544–546.

[Le Boudec and Thiran, 2001] Le Boudec, J. and Thiran, P. (2001). *Network Calculus*. Springer, LNCS # 2050.

[Lee, 1999] Lee, E. A. (1999). Embedded software – an agenda for research. Technical report, UCB ERL Memorandum M99/63.

[Lee, 2006] Lee, E. A. (2006). The future of embedded software. *ARTEMIS Conference, Graz, http://ptolemy.eecs.berkeley.edu/presentations/06/FutureOfEmbeddedSoftware_Lee_Graz.ppt*.

[Lee, 2007] Lee, E. A. (2007). Computing foundations and practice for cyber-physical systems: A preliminary report. Technical Report UCB/EECS-2007-72, EECS Department, University of California, Berkeley.

[Lee, 2005] Lee, E. A. (July, 2005). Absolutely positively on time. *IEEE Computer*.

[Lee and Messerschmitt, 1987] Lee, E. A. and Messerschmitt, D. (1987). Synchronous data flow. *Proc. of the IEEE, vol. 75*, pages 1235–1245.

[Lee et al., 2001] Lee, S., Ermedahl, A., and Min, S. (2001). An accurate instruction-level energy consumption model for embedded ROSC processors. *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–10.

[Leupers, 1997] Leupers, R. (1997). *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers.

[Leupers, 1999] Leupers, R. (1999). Exploiting conditional instructions in code generation for embedded VLIW processors. *Design, Automation and Test in Europe (DATE)*, pages 23–27.

[Leupers, 2000a] Leupers, R. (2000a). *Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools*. Kluwer Academic Publishers.

[Leupers, 2000b] Leupers, R. (2000b). Code selection for media processors with SIMD instructions. *Design, Automation and Test in Europe (DATE)*, pages 4–8.

[Leupers, 2000c] Leupers, R. (2000c). Instruction scheduling for clustered VLIW DSPs. *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 291–300.

[Leupers and David, 1998] Leupers, R. and David, F. (1998). A uniform optimization technique for offset assignment problems. *Int. Symp. on System Synthesis (ISSS)*, pages 3–8.

[Leupers and Marwedel, 1995] Leupers, R. and Marwedel, P. (1995). Time-constrained code compaction for DSPs. *Int. Symp. on System Synthesis (ISSS)*, pages 54–59.

[Leupers and Marwedel, 1996] Leupers, R. and Marwedel, P. (1996). Algorithms for address assignment in DSP code generation. *IEEE Int. Conf. on Computer-Aided Design (ICCAD)*, pages 109–112.

[Leupers and Marwedel, 1999] Leupers, R. and Marwedel, P. (1999). Function inlining under code size constraints for embedded processors. *IEEE Int. Conf. on Computer-Aided Design (ICCAD)*, pages 253–256.

[Leupers and Marwedel, 2001] Leupers, R. and Marwedel, P. (2001). *Retargetable Compiler Technology for Embedded Systems – Tools and Applications*. Kluwer Academic Publishers.

[Leveson, 1995] Leveson, N. (1995). *Safeware, System Safety and Computers*. Addison Wesley.

[Lewis et al., 2007] Lewis, J., Rashba, E., and Brophy, D. (2007). VHDL-2006-D3.0 Tutorial. *Tutorial at Design, Automation, and Test in Europe (DATE)*. *http://www.accellera.org/apps/group_public/download.php/934/date_vhdl_tutorial.pdf*.

[Liao et al., 1995a] Liao, S., Devadas, S., Keutzer, K., and Tijang, S. (1995a). Code optimization techniques for embedded DSP microprocessors. *32nd Design Automation Conference (DAC)*, pages 599–604.

[Liao et al., 1995b] Liao, S., Devadas, S., Keutzer, K., Tijang, S., and Wang, A. (1995b). Storage assignment to decrease code size. *Programming Language Design and Implementation (PLDI)*, pages 186–195.

[Liebisch and Jain, 1992] Liebisch, D. C. and Jain, A. (1992). Jessi common framework design management: the means to configuration and execution of the design process. In *Conf. on European Design Automation (EURO-DAC)*, pages 552–557. IEEE Computer Society Press.

[LIN Administration, 2010] LIN Administration (2010). Home page. *http://www.lin-subbus.org/*.

[Liu and Layland, 1973] Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery (JACM)*, pages 40–61.

[Liu, 2000] Liu, J. W. (2000). *Real-Time Systems*. Prentice Hall.

[Lohmann et al., 2009] Lohmann, D., Hofer, W., Schröder-Preikschat, W., and Spinczyk, O. (2009). CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *USENIX Annual Technical Conference*.

[Lohmann et al., 2006] Lohmann, D., Scheler, F., Schröder-Preikschat, W., and Spinczyk, O. (2006). PURE Embedded Operating Systems - CiAO. *Proc. International Workshop on Operating System Platforms for Embedded Real-Time Applications, (OSPERT)*.

[Lokuciejewski et al., 2009] Lokuciejewski, P., Gedikli, F., Marwedel, P., and Morik, K. (2009). Automatic WCET Reduction by Machine Learning Based Heuristics for Function Inlining. In *3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART)*, pages 1–15.

[Lokuciejewski and Marwedel, 2010] Lokuciejewski, P. and Marwedel, P. (2010). *WCET-aware Source Code and Assembly Level Optimization Techniques for Real-Time Systems*. Springer.

[Lorenz et al., 2004] Lorenz, M., Marwedel, P., Dräger, T., Fettweis, G., and Leupers, R. (2004). Compiler based exploration of DSP energy savings by SIMD operations. In *ASP-*

*DAC '04: Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, pages 838–841, Piscataway, NJ, USA. IEEE Press.

[Lorenz et al., 2002] Lorenz, M., Wehmeyer, L., Draeger, T., and Leupers, R. (2002). Energy aware compilation for DSPs with SIMD instructions. *LCTES/SCOPES*, pages 94–101.

[Lu et al., 2000] Lu, Y.-H., Chung, E.-Y., Šimunic, T., Benini, L., and De Micheli, G. (2000). Quantitative comparison of power management algorithms. In *Design, Automation and Test in Europe (DATE)*, pages 20–26.

[Machanik, 2002] Machanik, P. (2002). Approaches to addressing the memory wall. *Technical Report, November, Univ. Brisbane.*

[Macii et al., 2002] Macii, A., Benini, L., and Poncino, M. (2002). *Memory Design Techniques for Low Energy Embedded Systems*. Kluwer Academic Publishers.

[Macii, 2004] Macii, E., editor (2004). *Ultra low-power electronics and design*. Springer.

[Mahlke et al., 1992] Mahlke, S. A., Lin, D. C., Chen, W. Y., Hank, R. E., and Bringmann, R. A. (1992). Effective compiler support for predicated execution using the hyperblock. *25th annual Int. Symp. on Microarchitecture (MICRO)*, pages 45–54.

[Man, 2007] Man, H. D. (2007). From the heaven of software to the hell of nanoscale physics: an industry in transition... *Keynote, HiPEAC ACACES Summer School, L'Aquila.*

[Marian and Ma, 2007] Marian, N. and Ma, Y. (2007). Translation of Simulink models to component-based software models. *8th Int. Workshop on Research and Education in Mechatronics REM, http://seg.mci.sdu.dk/publications/Translation%20of%20Simulink%20Models%20to%20Component-based%20Software%20Models.pdf*, pages 262–267.

[Marongiu and Benini, 2009] Marongiu, A. and Benini, L. (2009). Efficient OpenMP support and extensions for MPSoCs with explicitly managed memory hierarchy. *Design, Automation and Test in Europe (DATE)*, pages 809–814.

[Martin and Müller, 2005] Martin, G. and Müller, W., editors (2005). *UML$^{TM}$ for SoC Design*. Springer.

[Martin et al., 2002] Martin, S. M., Flautner, K., Mudge, T., and Blaauw, D. (2002). Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 721–725, New York, NY, USA. ACM.

[Marwedel, 1990] Marwedel, P. (1990). A software system for the synthesis of computer structures and the generation of microcode (in German). *habilitation thesis, Universität Kiel, 1985, Reprint: Report Nr.356, CS Dept., TU Dortmund.*

[Marwedel, 2003] Marwedel, P. (2003). *Embedded System Design*. Kluwer Academic Publishers.

[Marwedel, 2005] Marwedel, P. (2005). Towards laying common grounds for embedded system design education. *ACM SIGBED Review*, pages 25–28.

[Marwedel, 2007] Marwedel, P. (2007). Memory-architecture aware compilation. *Tutorial, HiPEAC ACACES Summer School, L'Aquila, http://ls12-www.cs.tu-dortmund.de/publications/papers/2007-marwedel-acaces.zip.*

[Marwedel, 2008a] Marwedel, P. (2008a). 1st workshop on mapping of applications to MP-SoCs. *http://www.artist-embedded.org/artist/-map2mpsoc-2008-.html.*

[Marwedel, 2008b] Marwedel, P. (2008b). MIMOLA - a fully synthesizable language. *in: Prabhat Mishra, Nikil Dutt (Ed.): Processor Description Languages - Applications and Methodologies, Morgan Kaufmann*, pages 35–63.

[Marwedel, 2009a] Marwedel, P. (2009a). 2nd workshop on mapping of applications to MP-SoCs. *http://www.artist-embedded.org/artist/-map2mpsoc-2009-.html.*

[Marwedel, 2009b] Marwedel, P. (2009b). Mapping of applications to MPSoCs. *IP-Embedded Systems Conference, Grenoble, http://ls12-www.cs.tu-dortmund.de/publications/papers/2009-ip-esc-marwedel.pdf.*

[Marwedel and Goossens, 1995] Marwedel, P. and Goossens, G., editors (1995). *Code Generation for Embedded Processors*. Kluwer Academic Publishers.

[Marwedel and Schenk, 1993] Marwedel, P. and Schenk, W. (1993). Cooperation of synthesis, retargetable code generation and test generation in the MSS. *European Design and Test Conf. (EDAC-EUROASIC)*, pages 63–69.

[Marzano and Aarts, 2003] Marzano, S. and Aarts, E. (2003). *The New Everyday*. 010 Publishers.

[Marzario et al., 2004] Marzario, L., Lipari, G., Balbastre, P., and Crespo, A. (2004). IRIS: a new reclaiming algorithm for server-based real-time systems. *Real-Time Application Symposium (RTAS 04).*

[Massa, 2002] Massa, A. J. (2002). *Embedded Software Development with eCos*. Prentice Hall.

[MathWorks, 2010] MathWorks, T. (2010). Stateflow 7.3. *http://www.mathworks.com/products/stateflow.*

[M$^c$Gregor, 2002] M$^c$Gregor, I. (2002). The relationship between simulation and emulation. *Winter Simulation Conference*, pages 1683–1688.

[McLaughlin and Moore, 1998] McLaughlin, M. and Moore, A. (1998). Real-Time Extensions to UML. *http://www.ddj.com/184410749.*

[McNamee et al., 2001] McNamee, D., Walpole, J., Pu, C., Cowan, C., Krasic, C., Goel, A., Wagle, P., Consel, C., Muller, G., and Marlet, R. (2001). Specialization tools and techniques for systematic optimization of system software. *ACM Trans. Comput. Syst.*, 19(2):217–251.

[Meijer et al., 2010] Meijer, S., Nikolov, H., and Stefanov, T. (2010). Throughput modeling to evaluate process merging transformations in polyhedral process networks. *Design, Automation and Test in Europe (DATE).*

[Menard and Sentieys, 2002] Menard, D. and Sentieys, O. (2002). Automatic evaluation of the accuracy of fixed-point algorithms. *Design, Automation and Test in Europe (DATE)*, pages 529–535.

[Merkel and Bellosa, 2005] Merkel, A. and Bellosa, F. (2005). Event-driven thermal management in SMP systems. *Proceedings of the Second Workshop on Temperature-Aware Computer Systems (TACS'05).*

[Mermet et al., 1998] Mermet, J., Marwedel, P., Ramming, F. J., Newton, C., Borrione, D., and Lefaou, C. (1998). Three decades of hardware description languages in Europe. *Journal of*

*Electrical Engineering and Information Science*, 3:106pp.

[Mesa-Martinez et al., 2010] Mesa-Martinez, F. J., Ardestani, E. K., and Renau, J. (2010). Characterizing processor thermal behavior. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 193–204, New York, NY, USA. ACM.

[MHPCC, 2010] MHPCC, M. (2010). SP parallel programming workshop - message passing interface (MPI). *http://www.mhpcc.edu/training/workshop/mpi/MAIN.html*.

[Microsoft Inc., 2003] Microsoft Inc. (2003). Windows® embedded home. *http://www.microsoft.com/windowsembedded*.

[Mnemee project, 2010] Mnemee project (2010). Memory maNagEMEnt technology for adaptive and efficient design of Embedded systems. *http://www.mnemee.org*.

[Monteiro and van Leuken, 2010] Monteiro, J. and van Leuken, R., editors (2010). *Integrated circuit and system design: power and timing modeling, optimization and simulation : 19th international workshop, PATMOS 2009*. Springer LNCS 5953.

[MOST Cooperation, 2010] MOST Cooperation (2010). Home page. *http://www.mostcooperation.com/home*.

[MPI/RT forum, 2001] MPI/RT forum (2001). Document for the real-time message passing interface (MPI/RT-1.1). *http://www.mpirt.org/drafts/mpirt-report-18dec01.pdf*.

[Muchnick, 1997] Muchnick, S. S. (1997). *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, Inc.

[Mukherjee, 2008] Mukherjee, S. (2008). *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Müller, 2007] Müller, W. (2007). UML$^{TM}$ for SoC and embedded systems design. *DATE 2007 Friday Workshop, http://www.c-lab.de/uml-soc/uml-date07/date07-uml-workshop.pdf*.

[Müller et al., 2003] Müller, W., Rosenstiel, W., and Ruf, J. (2003). *SystemC – Methodologies and Applications*. Kluwer Academic Publications.

[National Research Council, 2001] National Research Council (2001). *Embedded, Everywhere*. National Academies Press.

[National Science Foundation, 2010] National Science Foundation (2010). Cyber-Physical Systems (CPS). *http://www.nsf.gov/pubs/2010/nsf10515/nsf10515.htm*.

[National Space-Based Positioning, Navigation, and Timing Coordination Office, 2010] National Space-Based Positioning, Navigation, and Timing Coordination Office (2010). Global positioning system. *http://www.gps.gov*.

[Neumann, 1995] Neumann, P. G. (1995). *Computer Related Risks*. Addison Wesley.

[Neumann, 2010] Neumann, P. G., editor (2010). *The risks digest, forum on the risks to the public in computers and related Systems. http://catless.ncl.ac.uk/risks*.

[Nguyen et al., 2005] Nguyen, N., Dominguez, A., and Barua, R. (2005). Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. *Int. Conf. on Compilers,*

*architectures and synthesis for embedded systems (CASES)*, pages 115–125.

[Niemann, 1998] Niemann, R. (1998). *Hardware/Software Co-Design for Data-Flow Dominated Embedded Systems*. Kluwer Academic Publishers.

[Nikolov et al., 2008] Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A., Polstra, S., Bose, R., Zissulescu, C., and Deprettere, E. (2008). Daedalus: toward composable multimedia MP-SoC design. In *45th annual Design Automation Conference (DAC)*, pages 574–579.

[Nilsen, 1998] Nilsen, K. (1998). Adding real-time capabilities to Java. *Commun. ACM*, 41(6):49–56.

[Northeast Sustainable Energy Association, 2010] Northeast Sustainable Energy Association (2010). Zero-energy building award. *http://zeroenergybuilding.org*.

[Novosel, 2009] Novosel, D. (2009). Timing the power grid. *http://www.pserc.wisc.edu/documents/general_information/presentations/smartr_grid_executive_forum/*.

[Nuzman et al., 2006] Nuzman, D., Rosen, I., and Zaks, A. (2006). Auto-vectorization of interleaved data for SIMD. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 132–143, New York, NY, USA. ACM.

[Object Management Group (OMG), 2003] Object Management Group (OMG) (2003). CORBA® basics. *http://www.omg.org/gettingstarted/corbafaq.htm*.

[Object Management Group (OMG), 2005a] Object Management Group (OMG) (2005a). Real-time CORBA specification, version 1.2, jan. 2005. *http://www.omg.org/cgi-bin/doc?formal/05-01-04.ps*.

[Object Management Group (OMG), 2005b] Object Management Group (OMG) (2005b). UML$^{TM}$ profile for schedulability, performance, and time specification, version 1.1. *http://www.omg.org/cgi-bin/doc?formal/05-01-02.pdf*.

[Object Management Group (OMG), 2008] Object Management Group (OMG) (2008). OMG systems modeling language (OMG SysML$^{TM}$). *http://www.omg.org/spec/SysML/1.1/changebar/PDF*.

[Object Management Group (OMG), 2009] Object Management Group (OMG) (2009). A UML$^{TM}$ profile for MARTE: Modeling and analysis of real-time embedded systems - 1.0. *http://www.omg.org/spec/MARTE/1.0/PDF*.

[Object Management Group (OMG), 2010a] Object Management Group (OMG) (2010a). Catalog of UML$^{TM}$ profile specifications. *http://www.omg.org/technology/documents/profile_catalog.htm*.

[Object Management Group (OMG), 2010b] Object Management Group (OMG) (2010b). Unified modeling language (tm) resource page. *http://www.uml.org*.

[O'Neill, 2006] O'Neill, A. (2006). Analog to digital types. *IEEE tv (for members only), http://www.ieee.org/portal/ieeetv/viewer.html?progId=81045*.

[Open SystemC Initiative, 2005] Open SystemC Initiative (2005). IEEE 1666 LRM. *http://www.systemc.org/downloads/lrm*.

[OpenMP Architecture Review Board, 2008] OpenMP Architecture Review Board (2008).

OpenMP application program interface. *http://www.openmp.org/mp-documents/spec30.pdf.*

[Oppenheim et al., 2009] Oppenheim, A. V., Schafer, R., and Buck, J. R. (2009). *Digital Signal Processing*. Pearson Higher Education.

[OSEK Group, 2004] OSEK Group (2004). OSEK/VDX - communication (version 3.0.3). *http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf.*

[OSEK Group, 2010] OSEK Group (2010). Home page. *http://www.osek-vdx.org.*

[Palkovic et al., 2002] Palkovic, M., Miranda, M., and Catthoor, F. (2002). Systematic power-performance trade-off in MPEG-4 by means of selective function inlining steered by address optimisation opportunities. *Design, Automation and Test in Europe (DATE)*, pages 1072–1079.

[Pan et al., 2010] Pan, S., Hu, Y., and Li, X. (2010). IVF: Characterizing the vulnerability of microprocessor structures to intermittent faults. *Design, Automation and Test in Europe (DATE).*

[Parker, 1992] Parker, K. P. (1992). *The Boundary Scan Handbook*. Kluwer Academic Press.

[Paulin and Knight, 1987] Paulin, P. and Knight, J. (1987). Force-directed scheduling in automatic data path synthesis. *24th annual Design Automation Conference (DAC).*

[Petri, 1962] Petri, C. A. (1962). Kommunikation mit Automaten. *Schriften des Rheinisch-Westfälischen Institutes für instrumentelle Mathematik an der Universität Bonn.*

[Pino and Lee, 1995] Pino, J. L. and Lee, E. A. (1995). Hierarchical static scheduling of dataflow graphs onto multiple processors. *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, pages 2643–2646.

[Pohl et al., 2005] Pohl, K., Böckle, G., and van der Linden, F. (2005). *Software Product Line Engineering*. Springer, ISBN-10: 3540289011.

[Popovici et al., 2010] Popovici, K., Rousseau, F., Jerraya, A. A., and Wolf, M. (2010). *Embedded Software Design and Programming of Multiprocessor System-on-Chip*. Springer.

[Potop-Butucaru et al., 2006] Potop-Butucaru, D., de Simone, R., and Talpin, J.-P. (2006). The synchronous hypothesis and synchronous languages. *In: R. Zurawski (ed.): Embedded Systems Handbook, CRC Press.*

[Press, 2003] Press, D. (2003). *Guidelines for Failure Mode and Effects Analysis for Automotive, Aerospace and General Manufacturing Industries*. CRC Press.

[Pyka et al., 2007] Pyka, R., Faßbach, C., Verma, M., Falk, H., and Marwedel, P. (2007). Operating system integrated energy aware scratchpad allocation strategies for multi-process applications. *Int. Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 41–50.

[Quilleré and Rajopadhye, 2000] Quilleré, F. and Rajopadhye, S. (2000). Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22:773–815.

[Radetzki, 2009] Radetzki, M., editor (2009). *Languages for Embedded Systems and their*

*Applications*. Springer.

[Ramamritham, 2002] Ramamritham, K. (2002). System support for real-time embedded systems. *In: Tutorial 1, 39th Design Automation Conference (DAC)*.

[Ramamritham et al., 1998] Ramamritham, K., Shen, C., Gonzalez, O., Sen, S., and Shirgurkar, S. B. (1998). Using Windows NT for real-time applications: Experimental observations and recommendations. *IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 102–111.

[Reisig, 1985] Reisig, W. (1985). *Petri nets*. Springer Verlag.

[Ren et al., 2006] Ren, G., Wu, P., and Padua, D. (2006). Optimizing data permutations for SIMD devices. *ACM SIGPLAN Notices*, 41(6):118–131.

[Riccobene et al., 2005] Riccobene, E., Scandurra, P., Rosti, A., and Bocchio, S. (2005). A UML$^{TM}$ 2.0 profile for SystemC: toward high-level SoC design. In *5th ACM Int. Conf. on Embedded Software (EMSOFT)*, pages 138–141.

[Rixner et al., 2000] Rixner, S., Dally, W. J., Khailany, B. J., Mattson, P. J., and Kapasi, U. J. (2000). Register organization for media processing. *6th High-Performance Computer Architecture (HPCA-6)*, pages 375–386.

[Ruggiero and Benini, 2008] Ruggiero, M. and Benini, L. (2008). Mapping task graphs to the CELL BE processor. *http://www.artist-embedded.org/docs/Events/2008/Map2MPSoC/Map2mpsoc-08-ruggiero.pdf*.

[Russell and Jacome, 1998] Russell, T. and Jacome, M. F. (1998). Software power estimation and optimization for high performance, 32-bit embedded processors. *Int. Conf. on Computer Design (ICCD)*, pages 328–333.

[Ryan, 1995] Ryan, M. (1995). Market focus – insight into markets that are making the news in EE Times. *EE Times (was available at http://eetimes.com/columns/mfocus95/mfocus11.html)*.

[Sangiovanni-Vincentelli, 2002] Sangiovanni-Vincentelli, A. (2002). The context for platform-based design. *IEEE Design & Test of Computers*, page 120.

[Schmitz et al., 2002] Schmitz, M., Al-Hashimi, B., and Eles, P. (2002). Energy-efficient mapping and scheduling for DVS enabled distributed embedded systems. *Design, Automation and Test in Europe (DATE)*, pages 514–521.

[SDL Forum Society, 2009] SDL Forum Society (2009). List of commercial tools. *http://www.sdl-forum.org/Tools/Commercial.htm*.

[SDL Forum Society, 2010] SDL Forum Society (2010). Home page. *http://www.sdl-forum.org*.

[Sha et al., 1990] Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Trans. on Computers*, pages 1175–1185.

[Shi and Brodersen, 2003] Shi, C. and Brodersen, R. (2003). An automated floating-point to fixed-point conversion methodology. *Int. Conf. on Audio Speed and Signal Processing (ICASSP)*, pages 529–532.

[Siemens, 2010] Siemens (2010). Simatic step 7 programming software. *http://www.automation.siemens.com/simatic/industriesoftware/html_76/products/step7.htm*.

[Sifakis, 2008] Sifakis, J. (2008). A notion of expressiveness for component-based design. *Workshop on Foundations and Applications of Component-based Design, ES-Week, http://www.artist-embedded.org/docs/Events/2008/Components/SLIDES/12-JosephSifakis-WFCD-ArtistDesign-Oct192008.pdf.*

[Simple Scalar LLC, 2004] Simple Scalar LLC (2004). Home page. *http://www.simplescalar.com.*

[Simunic et al., 2000] Simunic, T., Benini, L., Acquaviva, A., Glynn, P., and De Micheli, G. (2000). Energy efficient design of portable wireless devices. *Intern. Symp. on Low Power Electronics and Design (ISLPED)*, pages 49–54.

[Simunic et al., 2001] Simunic, T., Benini, L., Acquaviva, A., Glynn, P., and De Micheli, G. (2001). Dynamic voltage scaling and power management for portable systems. *Design Automation Conference (DAC)*, pages 524–529.

[Simunic et al., 1999] Simunic, T., Benini, L., and De Micheli, G. (1999). Cycle-accurate simulation of energy consumption in embedded systems. *Design Automation Conference (DAC)*, pages 876–872.

[Simunic-Rosing et al., 2007] Simunic-Rosing, T., Coskun, A. K., and Whisnant, K. (2007). Temperature aware task scheduling in MPSoCs. *Design, Automation and Test in Europe (DATE)*, pages 1659–1664.

[Sipser, 2006] Sipser, M. (2006). *Introduction to the Theory of Computation*. Thomson Course Technology, Parts One and Two.

[Sirocic and Marwedel, 2007a] Sirocic, B. and Marwedel, P. (2007a). Levi Flexray® simulation software. *http://ls12-www.cs.tu-dortmund.de/teaching/download/levi/download/leviFRP.zip.*

[Sirocic and Marwedel, 2007b] Sirocic, B. and Marwedel, P. (2007b). Levi KPN simulation software. *http://ls12-www.cs.tu-dortmund.de/teaching/download/levi/download/leviKPN.zip.*

[Sirocic and Marwedel, 2007c] Sirocic, B. and Marwedel, P. (2007c). Levi RTS simulation software. *http://ls12-www.cs.tu-dortmund.de/teaching/download/levi/download/leviRTS.zip.*

[Sirocic and Marwedel, 2007d] Sirocic, B. and Marwedel, P. (2007d). Levi TDD simulation software. *http://ls12-www.cs.tu-dortmund.de/teaching/download/levi/download/leviTDD.zip.*

[Skadron et al., 2009] Skadron, K., Stan, M. R., Ribando, R. J., Gurumurthi, S., Huang, W., Sankaranarayanan, K., Tarjan, D., Burr, J., Ghosh, S., Velusamy, S., and Link, G. (2009). Hotspot 5.0. *http://lava.cs.virginia.edu/HotSpot/index.htm.*

[Smith and Nair, 2005] Smith, J. J. and Nair, R. (2005). *Virtual Machines: Versatile Platforms For Systems And Processes*. Morgan Kaufmann Publishers.

[Society for Display Technology, 2003] Society for Display Technology (2003). Home page. *http://www.sid.org.*

[Sprint Consortium, 2008] Sprint Consortium (2008). Open SoC design platform for reuse and integration of IPs. *http://www.sprint-project.net.*

[Stallings, 2009] Stallings, W. (2009). *Operating Systems: Internals and Design Principles*. Prentice Hall.

[Stankovic and Ramamritham, 1991] Stankovic, J. and Ramamritham, K. (1991). The Spring kernel: a new paradigm for real-time systems. *IEEE Software*, 8:62–72.

[Stankovic et al., 1998] Stankovic, J., Spuri, M., Ramamritham, K., and Buttazzo, G. (1998). *Deadline Scheduling for Real-Time Systems, EDF and related algorithms*. Kluwer Academic Publishers.

[Steinke, 2003] Steinke, S. (2003). *Analysis of the potential for saving energy in embedded systems through energy-aware compilation (in German)*. PhD thesis, TU Dortmund, *http://hdl.handle.net/2003/2769*.

[Steinke et al., 2002a] Steinke, S., Grunwald, N., Wehmeyer, L., Banakar, R., Balakrishnan, M., and Marwedel, P. (2002a). Reducing energy consumption by dynamic copying of instructions onto onchip memory. *Int. Symp. on System Synthesis (ISSS)*, pages 213–218.

[Steinke et al., 2001] Steinke, S., Knauer, M., Wehmeyer, L., and Marwedel, P. (2001). An accurate and fine grain instruction-level energy model supporting software optimizations. *Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*.

[Steinke et al., 2002b] Steinke, S., Wehmeyer, L., Lee, B.-S., and Marwedel, P. (2002b). Assigning program and data objects to scratchpad for energy reduction. *Design, Automation and Test in Europe (DATE)*, pages 409–417.

[Stiller, 2000] Stiller, A. (2000). New processors (in German). *c't*, 22:52.

[Storey, 1996] Storey, N. (1996). *Safety-critical Computer Systems*. Addison Wesley.

[Stritter and Gunter, 1979] Stritter, E. and Gunter, T. (1979). Microprocessor architecture for a changing world: The Motorola 68000. *IEEE Computer*, 12:43–52.

[Stuijk, 2007] Stuijk, S. (2007). *Predictable Mapping of Streaming Applications on Multiprocessors*. Dissertation, TU Eindhoven.

[Sudarsanam et al., 1997] Sudarsanam, A., Liao, S., and Devadas, S. (1997). Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. *Design Automation Conference (DAC)*, pages 287–292.

[Sudarsanam and Malik, 1995] Sudarsanam, A. and Malik, S. (1995). Memory bank and register allocation in software synthesis for ASIPs. *Intern. Conf. on Computer-Aided Design (ICCAD)*, pages 388–392.

[Sun, 2010] Sun (2010). Java technology concept map. *http://java.sun.com/new2java/javamap/Java_Technology_Concept_Map.pdf*.

[Sutherland, 2003] Sutherland, S. (2003). An overview of SystemVerilog 3.1. *EEdesign, May, available at http://www.eetimes.com/news/design/features/showArticle.jhtml?articleID=16501063*.

[Synopsys, 2010] Synopsys (2010). System studio. *http://www.synopsys.com/apps/docs/pdfs/ip/system_studio_ds.pdf*.

[SYSGO AG, 2010] SYSGO AG (2010). PikeOS RTOS and Virtualization Concept. *http://www.sysgo.com.*

[SystemC, 2010] SystemC (2010). Home page. *http://www.SystemC.org.*

[Takada, 2001] Takada, H. (2001). Real-time operating system for embedded systems. *In: M. Imai and N. Yoshida (eds.): Tutorial 2 – Software Development Methods for Embedded Systems, Asia South-Pacific Design Automation Conference (ASP-DAC).*

[Tan et al., 2003] Tan, T. K., Raghunathan, A., and Jha, N. K. (2003). Software architectural transformations: A new approach to low energy embedded software. *Design, Automation and Test in Europe (DATE)*, pages 11046–11051.

[Tanenbaum, 2001] Tanenbaum, A. (2001). *Modern Operating Systems.* Prentice Hall.

[Teich, 1997] Teich, J. (1997). *Digitale Hardware/Software-Systeme.* Springer.

[Teich et al., 1999] Teich, J., Zitzler, E., and Bhattacharyya, S. (1999). 3D exploration of software schedules for DSP algorithms. *7th Int. Symp. on Hardware/Software Codesign (CODES)*, pages 168–172.

[Tensilica Inc., 2010] Tensilica Inc. (2010). Home page. *http://www.tensilica.com.*

[Tewari, 2001] Tewari, A. (2001). *Modern Control Design with MATLAB and SIMULINK.* John Wiley and Sons Ltd.

[The Dobelle Institute, 2003] The Dobelle Institute (2003). Home page. *http://www.dobelle.com (no longer accessible).*

[The MathWorks Inc., 2010] The MathWorks Inc. (2010). Simulink - simulation and model-based design. *http://www.mathworks.com/products/simulink.*

[Thesing, 2004] Thesing, S. (2004). *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models.* Pirrot Verlag.

[Thiébaut, 1995] Thiébaut, D. (1995). Parallel programming in C for the transputer. *http://cs.smith.edu/~thiebaut/transputer/descript.html.*

[Thiele, 2006a] Thiele, L. (2006a). Design space exploration of embedded systems. *Artist Spring School on Embedded Systems, Xi-an, http://www.artist-embedded.org/docs/Events/2006/ChinaSchool/4_DesignSpaceExploration.pdf.*

[Thiele, 2006b] Thiele, L. (2006b). Performance analysis of distributed embedded systems. *In: R. Zurawski (Hrg.): Embedded Systems Handbook, CRC Press, 2006.*

[Thiele, L. et al., 2009] Thiele, L. et al. (2009). SHAPES @ TIK. *http://www.tik.ee.ethz.ch/~shapes/dol.html.*

[Thoen and Catthoor, 2000] Thoen, F. and Catthoor, F. (2000). *Modelling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems.* Kluwer Academic Publishers.

[Tiwari et al., 1994] Tiwari, V., Malik, S., and Wolfe, A. (1994). Power analysis of embedded software: A first step towards software power minimization. *IEEE Trans. On VLSI Systems*, pages 437–445.

[Tjiang, 1993] Tjiang, W.-K. (1993). An olive twig. *Technical Report, Synopsys.*

[Trimaran, 2010] Trimaran (2010). An infrastructure for research in backend compilation and architecture exploration. *http://www.trimaran.org.*

[TriQuint Semiconductor Inc., 2010] TriQuint Semiconductor Inc. (2010). FAQ 11: What is the MTBF for gallium arsenide devices? *http://www.triquint.com/company/quality/faqs/faq_11.cfm.*

[Tsai and Yang, 1995] Tsai, J. and Yang, S. J. H. (1995). *Monitoring and Debugging of Distributed Real-Time Systems.* IEEE Computer Society Press.

[Udayakumaran et al., 2006] Udayakumaran, S., Dominguez, A., and Barua, R. (2006). Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions in Embedded Computing Systems*, V:472–511.

[University of Cambridge, 2010] University of Cambridge (2010). HOL4. *http://hol.sourceforge.net.*

[UPnP Forum, 2010] UPnP Forum (2010). UPnP $^{TM}$ resources. *http://www.upnp.org/resources/default.asp.*

[V-Modell XT Authors, 2010] V-Modell XT Authors (2010). V-Modell XT Gesamt 1.3. *http://v-modell.iabg.de/dmdocuments/V-Modell-XT-Gesamt-Englisch-V1.3.pdf.*

[Vaandrager, 1998] Vaandrager, F. (1998). Lectures on embedded systems. *in Rozenberg, Vaandrager (eds), LNCS, Vol. 1494.*

[Vahid, 1995] Vahid, F. (1995). Procedure exlining. *Int. Symp. on System Synthesis (ISSS)*, pages 84–89.

[Vahid, 2002] Vahid, F. (2002). *Embedded System Design.* John Wiley& Sons.

[Verachtert, 2008] Verachtert, W. (2008). Introduction to parallelism. *Tutorial at Design, Automation, and Test in Europe (DATE).*

[Verma and Marwedel, 2004] Verma, M. and Marwedel, P. (2004). Dynamic overlay of scratchpad memory for energy minimization. *8th IEEE/ACM Int. Conf. on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 104–109.

[Verma et al., 2005] Verma, M., Petzold, K., Wehmeyer, L., Falk, H., and Marwedel, P. (2005). Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *IEEE 3rd Workshop on Embedded System for Real-Time Multimedia (ESTIMedia)*, pages 115–120.

[Vladimirescu, 1987] Vladimirescu, A. (1987). SPICE user's guide. *Northwest Laboratory for Integrated Systems, Seattle.*

[Vogels and Gielen, 2003] Vogels, M. and Gielen, G. (2003). Figure of merit based selection of A/D converters. *Design, Automation and Test in Europe (DATE)*, pages 1190–1191.

[Wandeler and Thiele, 2006] Wandeler, E. and Thiele, L. (2006). Real-Time Calculus (RTC) Toolbox.

[Wedde and Lind, 1998] Wedde, H. and Lind, J. (1998). Integration of task scheduling and file services in the safety-critical system MELODY. *EUROMICRO '98 Workshop on Real-Time Systems, IEEE Computer Society Press*, pages 18–25.

[Wegener, 2000] Wegener, I. (2000). *Branching programs and binary decision diagrams –*

*Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications.

[Wehmeyer and Marwedel, 2006] Wehmeyer, L. and Marwedel, P. (2006). *Fast, Efficient and Predictable Memory Accesses*. Springer.

[Weiser, 2003] Weiser, M. (2003). Ubiquitous computing. *http://www.ubiq.com/hypertext/weiser/UbiHome.html*.

[Wellings, 2004] Wellings, A. (2004). *Concurrent and Real-Time Programming in Java*. Wiley.

[Weste et al., 2000] Weste, N. H. H., Eshraghian, K., Michael, S., Michael, J. S., and Smith, J. S. (2000). *Principles of CMOS VLSI Design: A Systems Perspective*. Addision-Wesley.

[Wikipedia, 2010] Wikipedia (2010). Structured systems analysis and design method. *http://en.wikipedia.org/wiki/Structured_Systems_Analysis_and_Design_Methodology*.

[Wilhelm, 2006] Wilhelm, R. (2006). Determining bounds on execution times. *In: R. Zurawski (Ed.): Embedded Systems Handbook, CRC Press, 2006*.

[Willems et al., 1997] Willems, M., Bürsgens, V., Keding, H., Grötker, T., and Meyr, H. (1997). System level fixed-point design based on an interpolative approach. *Design Automation Conference (DAC)*, pages 293–298.

[Wilton and Jouppi, 1996] Wilton, S. and Jouppi, N. (1996). CACTI: An enhanced access and cycle time model. *Int. Journal on Solid State Circuits*, 31(5):677–688.

[Wind River, 2010a] Wind River (2010a). VxWorks. *http://www.windriver.com/products/vxworks*.

[Wind River, 2010b] Wind River (2010b). Web pages. *http://www.windriver.com*.

[Winkler, 2002] Winkler, J. (2002). The CHILL homepage. *http://psc.informatik.uni-jena.de/languages/chill/chill.htm*.

[Wolf, 2001] Wolf, W. (2001). *Computers as Components*. Morgan Kaufmann Publishers.

[Wolsey, 1998] Wolsey, L. (1998). *Integer Programming*. Jon Wiley & Sons.

[Wong et al., 2001] Wong, C., Marchal, P., Yang, P., Prayati, A., Catthoor, F., Lauwereins, R., Verkest, D., and Man, H. D. (2001). Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform. *9th Int. Symp. on Hardware/Software Codesign (CODES)*, pages 170–177.

[ws4d, 2010] ws4d (2010). Web services for devices. *http://www.ws4d.org*.

[Xilinx, 2008] Xilinx (2008). MicroBlaze processor reference guide. *http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf*.

[Xilinx, 2009] Xilinx (2009). Device reliability report - second quarter 2009. *http://www.xilinx.com/support/documentation/user_guides/ug116.pdf*.

[Xilinx, 2009] Xilinx (May, 2009). Virtex-5 user guide, v 4.7. *http://www.xilinx.com/support/documentation/user_guides/ug190.pdf*.

[Xilinx, 2007] Xilinx (Nov., 2007). Virtex-II Platform User Guide, V 2.2. *http://www.xilinx.com/support/documentation/user_guides/ug002.pdf*.

[XMOS Ltd., 2010] XMOS Ltd. (2010). Home page. *http://www.xmos.com/.*

[Xu and Parnas, 1993] Xu, J. and Parnas, D. L. (1993). On satisfying timing constraints in hard real-time systems. *IEEE Transactions on Software Engineering*, 19:70–84.

[Xu et al., 2009] Xu, Q., Huang, L., and Yuan, F. (2009). Lifetime reliability-aware task allocation and scheduling for MPSoC platforms. *Design, Automation and Test in Europe (DATE)*, pages 51–56.

[Xue, 2000] Xue, J. (2000). *Loop tiling for parallelism.* Kluwer Academic Publishers.

[Young, 1982] Young, S. (1982). *Real Time Languages – design and development.* Ellis Horwood.

[Zhuo et al., 2010] Zhuo, C., Sylvester, D., and Blaauw, D. (2010). Process variation and temperature-aware reliability management. *Design, Automation and Test in Europe (DATE).*

[Zurawski, 2006] Zurawski, R., editor (2006). *Embedded Systems Handbook.* CRC Press.

# 国际视野　科技前沿

## 国际信息工程先进技术译丛

《嵌入式系统设计 —— 嵌入式信息物理系统基础》（原书第2版）
《纳米封装 —— 纳米技术与电子封装》
《内容分发网络》
《全面的功能验证：完整的工业流程》
《无线Mesh网络架构与协议》
《UMTS蜂窝系统的QoS与QoE管理》
《半导体制造与过程控制基础》
《WCDWA原理与开发设计》
《下一代移动系统：3G/B3G》
《IMS:IP多媒体概念和服务》（原书第2版）
《下一代无线系统与网络》
《深入浅出UMTS无线网络建模、
规划与自动优化：理论与实践》
《HSDPA/HSUPA技术与系统设计——第三代移动
通信系统宽带无线接入》
《无线传感器及元器件：网络、设计与应用》
《印制电路板——设计、制造、装配与测试》
《IPTV与网络视频：拓展广播电视的应用范围》
《多电压CMOS电路设计》
《微电子技术原理、设计与应用》
《蜂窝网络高级规划与优化2G/2.5G/3G/…向4G的演进》
《基于蜂窝系统的IMS——融合电信领域的VoIP演进》
《无线网络中的合作原理与应用》
《电生理学方法与仪器入门》
《移动电视：DVB-H、DMB、3G系统和富媒体应用》
《环境网络：支持下一代无线业务的多域协同网络》
《基于射频工程的UMTS空中接口设计与网络运行》
《未来UMTS的体系结构与业务平台：全IP的3G CDMA网络》
《UMTS-HSDPA系统的TCP性能》
《宽带无线通信中的空时编码》
《数字图像处理》（原书第4版）
《基于4G系统的移动服务技术》
《吉规模集成电路互连工艺及设计》
《高性能微处理器电路设计》




上架指导　工业技术/自动化/嵌入式技术

ISBN 978-7-111-41255-7　定价：79.90元

[General Information]
书名=嵌入式系统设计：嵌入式信息物理系统基础（原书第2版）
页数=279
SS号=13205332